Calhoun

Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1987

Adaptive identification by systolic arrays.

Willis, Paul A.; Cristi, Roberto.

http://hdl.handle.net/10945/22187

NAVAL POSTGRADUATE SCHOOL
Monterey, California

ADAPTIVE IDENTIFICATION
BY SYSTOLIC ARRAYS

by

Paul A. Willis

December 1987

Thesis Advisor   Roberto Cristi

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) Code 33	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

ADAPTIVE IDENTIFICATION BY SYSTOLIC ARRAYS

12. PERSONAL AUTHOR(S)
Willis, Paul A.

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 December	15. PAGE COUNT 68

16. SUPPLEMENTARY NOTATION

17	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Systolic Arrays, Adaptive Identification, Parallel Processing, Cordic Algorithm

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis is concerned with the implementation of an adaptive identification algorithm using parallel processing and systolic arrays. In particular, discrete samples of input and output data of a system with uncertain characteristics are used to determine the parameters of its model. The identification algorithm is based on recursive least squares, QR decomposition, and block processing techniques with covariance resetting. Along similar lines as previous approaches, the identification process is based on the use of Givens rotations. This approach uses the Cordic technique for improved numerical efficiency in performing the rotations. Additionally, floating point and fixed point arithmetic implementations are compared.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Roberto Cristi	22b TELEPHONE (Include Area Code) (408) 646-2223 22c OFFICE SYMBOL Code 62Cx

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted. All other editions are obsolete   SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-24.

Adaptive Identification
by Systolic Arrays

by

Paul A. Willis
Lieutenant, United States Navy
B.S.E.E., Purdue University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1987

ABSTRACT

This thesis is concerned with the implementation of an adaptive identification algorithm using parallel processing and systolic arrays. In particular, discrete samples of input and output data of a system with uncertain characteristics are used to determine the parameters of its model. The identification algorithm is based on recursive least squares, QR decomposition, and block processing techniques with covariance resetting. Along similar lines as previous approaches, the identification process is based on the use of Givens rotations. This approach uses the Cordic technique for improved numerical efficiency in performing the rotations. Additionally, floating point and fixed point arithmetic implementations are compared.

3

TABLE OF CONTENTS

4

5

LIST OF TABLES

6

I. INTRODUCTION

A.   BACKGROUND

The problem of adaptively controlling systems with uncertain characteristics depends largely on identification of the unknown system parameters. The ability to accurately and quickly estimate these parameters, then, is of primary importance. The state of computer development today has made this estimation possible in real time.

The goal of parameter estimation is to best fit an appropriate model to the input-output data of the plant under investigation. This immediately leaves us with two basic and distinct problems: the choice of a parameter model and the choice of an estimation algorithm.

We desire to select a parameter model which relates input and output data by means of weighted parameters, in the form

$$y(t) = \varphi^T(t)\,\theta + v(t) \tag{1.1}$$

with $y(t)$ and $\varphi^T(t)$ representing the output and signals respectively available for measurements, and where θ is an array of unknown parameters to be determined. The term $v(t)$ represents noise or other modeling errors. Though numerous choices of model structures exist, linear models still remain the most desirable due to their simplicity and the considerable amount of theory developed to analyze them.

Secondly, we need to choose an appropriate estimation algorithm. Again, several possibilities exist, but the most effective in terms of converging is the recursive least squares algorithm [Ref. 1: pp. 49-68]. This algorithm is implementable using on line matrix manipulations, and the technique is based upon on line solution of a system of linear equations.

The major drawback of recursive least squares is the computational complexity involved. The size of the matrices grows with the complexity of the system to be modeled. Although available microprocessors are effective for low order systems and slow sampling rates, more complex problem require improved capabilities. These capabilities are provided by systolic arrays built using VLSI technology.

This research analyzes the on line recursive least squares identification algorithm. This algorithm processes blocks of data, using values gained from the previous block as initializing values, in a method known as block processing. This block processing technique is based on QR decomposition, discussed in the next chapter.

B.   IMPLEMENTATION CONSIDERATIONS

The particular computing structure we are considering is based on the systolic array (or wavefront array). The systolic array consists of an array of individual processing cells, each provided with a local memory and processing unit of its own, connected in such a way that each cell communicates only with its nearest neighbors. The array is designed so that data is continuously clocked or pumped throughout in a rhythmic fashion; hence the name "systolic." The cells are simple in that they are required to perform only basic mathematical functions on the data received from neighboring cells. Special purpose hardware incorporating systolic arrays can be built using VLSI technology. The advantage with this technique is the fact that a complex operation is performed by several processors at a time, thus increasing the throughput of data.

Previous authors [Refs. 2,3: pp.29 - 37, pp.255 - 274] have presented parameter estimation algorithms using systolic arrays. The general idea has been to solve a system of linear equations in two stages: first, by triangularization of the matrix of coefficients, and second, solving by successive substitution. The previous algorithms have all been similar in that they used a triangular systolic array to triangularize the matrix, and a linear systolic array configuration to solve for the parameters. It turns out that the arrays for the triangular and linear sections are different, and furthermore the linear section requires operations such as divisions which are hardly implementable by simple processor operations.

In this thesis we present an algorithm which is based on two identical triangular sections. It is characterized by the fact that only orthogonal operations are involved, thus making the algorithm numerically more stable, and easily implementable by simple shift and add operations. Also, one of the advantages is that only two different types of cells are necessary (as compared to four in the previous implementations), resulting in a simplification in the manufacturing process. These cells perform different functions from those used by the above referenced authors. The Cordic technique is used by the cells to perform vector rotations, resulting in improved numerical efficiency. Though more total cells are required in this implementation, the cost of additional cells in a VLSI scheme is considered to be minimal.

The use of fixed point versus floating point arithmetic is considered during this investigation. Because fixed point operations are based on simple shift functions and finite registers, which are simple to implement, it seemed advantageous to use fixed point values. However, since input and output data do not naturally appear as integer values, there was concern over loss of accuracy due to necessary scaling and truncation. With proper choice of scaling factors, it will be shown that the integer methods perform as well as floating point.

This thesis is divided as follows: Chapter II discusses the methods of system identification, i.e. solution of systems of linear equations, QR decomposition, and recursive least squares; Chapter III discusses the Cordic technique and the implementation of the systolic arrays. Chapter IV discusses the simulation results, and Chapter V draws the final conclusions. A listing of the program used to simulate the systolic arrays is found in the appendix.

II. SYSTEM IDENTIFICATION METHODS

A. LINEAR SYSTEM MODELING

The first step in any parameter identification process is to model the system in mathematical terms. To this end, consider a general system with a single input $u(t)$ and single output $y(t)$, as shown in Figure 2.1.
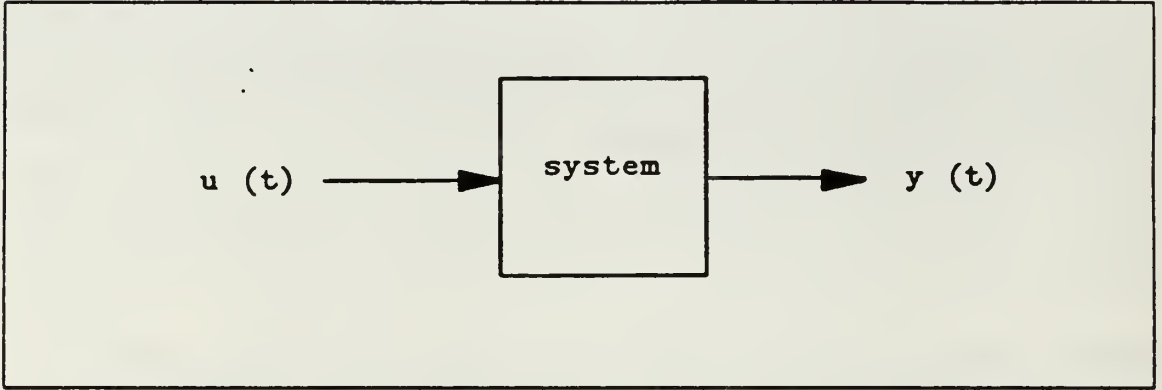
Figure 2.1   Simple Linear System.

This system can be described by the linear, constant coefficient difference equation

$$y(t) + a_1 y(t\text{-}1) + \ldots + a_n y(t\text{-}n) = b_1 u(t\text{-}1) + \ldots + b_m u(t\text{-}m) + v(t) \qquad (2.1)$$

where the a_i's and the b_i's are real constants, and the equation (and system) is of nth order. The $v(t)$ term denotes a noise variable. Equation (2.1) represents a class of discrete systems, known as *recursive* systems because the output depends not only on the input but on the previous output values also.

An alternative way to represent the above equation is by defining the parameter vector as

$$\theta^T = [a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m] \qquad (2.2)$$

and the regression vector of input-output data as

$$\varphi^T(t) = [\,\text{-}y(t\text{-}1), \ldots, \text{-}y(t\text{-}n), u(t\text{-}1), \ldots, u(t\text{-}m)] \qquad (2.3)$$

12

Now, using (2.2) and (2.3), we can rewrite (2.1) as

$$y(t) = \varphi^T(t)\,\theta + v(t) \tag{2.4}$$

The vector θ contains the parameters which we want to determine. This can be obtained by sampling the system at some sample frequency (say, ω_s) and accumulating a numerical sequence that describes the system at discrete time intervals. We can express this in the form of a system of linear equations by considering the sequence at several instants of time. Ideally, if the number of samples of $u(t)$ and $y(t)$ (i.e. the number of equations) equals the number of unknowns $(m + n)$, we should be able to solve exactly for θ. However, it is normally the case that the number of equations is greater than the number of unknowns. This system may or may not have a solution. Ideally, in the noiseless case $(v(t) = 0)$ a solution exists, regardless of the number of equations. However, since noise and numerical errors are present, we look for the least squares solution of the system of equations, i.e. for the solution which minimizes the error

$$\varepsilon(\theta) = \|\, A\theta - b\,\|^2 \tag{2.5}$$

This always exists, although it might not be unique.

B.   SOLUTION OF SYSTEMS OF LINEAR EQUATIONS

In the previous section we saw how the linear system could be modeled by (2.4). When numerous samples are taken, we end up with a system of linear equations, such as

$$
\begin{aligned}
y(t) &= \varphi^T(t)\,\theta + v(t) \\
y(t\text{-}1) &= \varphi^T(t\text{-}1)\,\theta + v(t\text{-}1) \\
y(t\text{-}2) &= \varphi^T(t\text{-}2)\,\theta + v(t\text{-}2) \\
&\quad\vdots
\end{aligned}
$$

This system of equations can be expressed in the form

$$A\theta = \underline{b} \tag{2.6}$$

13

where $A \in \mathcal{R}^{m \times n}$, $\theta \in \mathcal{R}^n$, $\underline{b} \in \mathcal{R}^m$. Now, when m = n, and A is full rank and invertible, we can solve uniquely for θ as

$$\theta = A^{-1}\underline{b} \tag{2.7}$$

However, in general we find that m > n (i.e. more equations than unknowns). In this instance, the solution of (2.7) is defined in the least squares sense as

$$\theta_s \quad \text{where} \quad \|A\theta_s - \underline{b}\| = \min_\theta \|A\theta - \underline{b}\| \tag{2.8}$$

Although (2.8) can be solved by pseudoinverse, this technique is not attractive due to the presence of matrix inversion. An alternative way is to triangularize A in (2.6) and then solve for θ by successive back substitutions. This, of course, is the basis of Gaussian elimination techniques.

But now we face the dilemma of triangularizing an array when m > n. The solution is to triangularize the upper part of the A matrix, leaving one or more rows of zeros at the bottom as a result. This will permit us to solve for θ in the *least squares sense*, as indicated in (2.8). This is known as QR decomposition. (For review of least squares estimation in statistical theory, the reader is referred to the literature.)

C. QR DECOMPOSITION

A means to solve for the least-squares solution of a system of linear equations is provided by the QR decomposition of a matrix [Ref. 4: pp. 209-215]. Consider again Equation (2.6) with m > n, where $\theta = \theta_s$ (i.e. least squares sense). We can factor the m × n matrix A as

$$A = Q\underline{R} \tag{2.9}$$

where Q is a m × n orthogonal matrix such that $QQ^T = I$, and $\underline{R}$ is defined as

$$\underline{R} = \left| \begin{array}{c} R \\ \hline 0 \end{array} \right| \tag{2.10}$$

where R is an n × n upper triangular matrix, and 0 represents a zero matrix. Now we can rewrite (2.6) as

$$\left| \begin{array}{c} R \\ \hline 0 \end{array} \right| \theta = Q^T \underline{b} = \left| \begin{array}{c} \beta_1 \\ \hline \beta_2 \end{array} \right| \tag{2.11}$$

And finally, from (2.11), it follows that

$$R \theta = \beta_1 \tag{2.12}$$

Now, the solution of (2.6) in the least squares sense is the same as the solution of n equations and n unknowns in (2.12).

Proof. Consider the set of equations

$$\underline{e} = A\theta - \underline{b} = QR\theta - \underline{b}$$

where e represents the least squares error vector. Then,

$$\underline{e}^T \underline{e} = (\underline{b}^T - \theta^T A^T)(A\theta - \underline{b})$$
$$Q^T \underline{e} = R\theta - Q^T \underline{b} = R\theta - \beta$$

Then, we have

$$\underline{e}^T \underline{e} = \underline{e}^T QQ^T \underline{e} = \| R\theta - \beta \|^2$$

$$= \left\| \left| \begin{array}{c} R \\ \hline 0 \end{array} \right| \theta - \left| \begin{array}{c} \beta_1 \\ \hline \beta_2 \end{array} \right| \right\|^2$$

$$= \left\| \left| \begin{array}{c} R\theta - \beta_1 \\ -\beta_2 \end{array} \right| \right\|^2$$

$$= \| R\theta - \beta_1 \|^2 + \| \beta_2 \|^2$$

We see that β_2 is independent of θ. Therefore, $\|e\|^2$ is minimized when

$$\theta_{ls} = R^{-1}\beta_1 \quad \text{or} \quad R\theta_{ls} = \beta_1 \quad \bullet$$

15

Based upon these results, we see that we can now solve (2.12) for θ by back substitution.

There are a number of methods available that can be used to compute the upper triangular matrix R. In particular, the Givens rotation [Ref. 5: pp. 497 - 499] is attractive for systolic array implementation since it is based on manipulation of pairs of adjacent rows, thus requiring minimum broadcasting of data. Cordic techniques are used to implement these rotations, and are discussed in the next chapter.

D.   RECURSIVE LEAST SQUARES

The problem now is to determine a recursive algorithm that can be used to estimate the parameter θ in the model

$$y(t) = \varphi^T(t)\,\theta \tag{2.13}$$

We desire to estimate θ from a set of measured data $y(t)$ and $\varphi(t)$. In the least squares sense we choose this estimate by best fitting our model to the available data. That is, we wish to minimize ε, where

$$|\varepsilon| = |y(t) - \varphi^T(t)\,\hat{\theta}| \tag{2.14}$$

$\hat{\theta}$ refers to the estimate of θ at any time t. In particular $\hat{\theta}_t$ satisfies the equations

$$\begin{vmatrix} \phi^T(t-1) \\ \cdot \\ \phi^T(0) \end{vmatrix} \hat{\theta}_t = \begin{vmatrix} y(t-1) \\ \cdot \\ y(0) \end{vmatrix} \rightarrow A(t-1)\hat{\theta}_t = y(t-1) \tag{2.15}$$

It is possible to show that $\hat{\theta}_t$ can be recursively computed as [Ref. 6: pp. 17 - 22]

$$\hat{\theta}_{t+1} = \hat{\theta}_t + P(t\text{-}1)\,\frac{\varphi(t)\,(y(t) - \hat{\theta}_t^T\,\varphi(t))}{1 + \varphi^T(t)\,P(t\text{-}1)\,\varphi(t)} \tag{2.16}$$

where $P(t) = (A(t)^T A(t))^{-1}$ satisfies the recursion

$$P(t) = P(t\text{-}1) - P(t\text{-}1)\,\frac{\varphi(t)\,\varphi^T(t)\,P(t\text{-}1)}{1 + \varphi^T(t)\,P(t\text{-}1)\,\varphi(t)} \tag{2.17}$$

16

To this point, we have ignored the problem of the existence of the solution. If A(t) is singular, then P(t) does not exist. Furthermore, we can not initialize A(-1) = 0 because this would leave P(-1) undefined. The solution is to incorporate initial conditions into A(t) so that P(t) can be computed at each t. Choose $A(-1) = \sigma_0 I$, where $\sigma_0 > 0$ is some arbitrary constant, and I is the identity matrix of appropriate dimensions. Then, by algebraic manipulation, Equation (2.15) becomes

$$
\begin{vmatrix} \phi^T(t-1) \\ \cdot \\ \phi^T(0) \\ \overline{} \\ A(-1) \end{vmatrix} \hat{\theta}_t = \begin{vmatrix} y(t-1) \\ \cdot \\ y(0) \\ \overline{} \\ A(-1)\hat{\theta}_0 \end{vmatrix}
\tag{2.18}
$$

The solution to (2.18) can be made fully recursive using (2.16) and (2.17) and appropriate initial conditions.

E.   BLOCK PROCESSING AND COVARIANCE RESETTING

In the previous section, an algorithm was described that allows us to estimate the parameter θ recursively. However, note that from the definition of P(t) (also known as the covariance matrix)

$$
P(t) = (A^T(t)\, A(t))^{-1} = (\sigma_0 I + \sum_{i=0}^{t} \varphi(i)\, \varphi^T(i)\,)^{-1}
\tag{2.19}
$$

that P(t) → 0 as t → ∞ [Ref. 6: p. 21]. Therefore, the algorithm loses sensitivity as t increases, and later values of θ may not be as accurate as earlier values, especially if our model changes with time. There are two possible remedies to this problem: the use of a "forgetting factor", or the block processing approach.

The forgetting factor minimizes the error

$$
\left|\left| \epsilon(t) \right|\right|^2 = \sum_{k=0}^{t} \lambda^{t-k}\, (y(k) - \phi^T(k)\hat{\theta}_{t+1})^2 + \sigma_0^2 \left|\left| \hat{\theta}_{t+1} - \theta_0 \right|\right|^2
\tag{2.20}
$$

where $0 < \lambda < 1$ is the forgetting factor. This has the effect of assigning a higher weight to more recent data.

The block processing approach, on the other hand, divides the time set into segments of equal and fixed length N as in Figure 2.2. At the end of each time
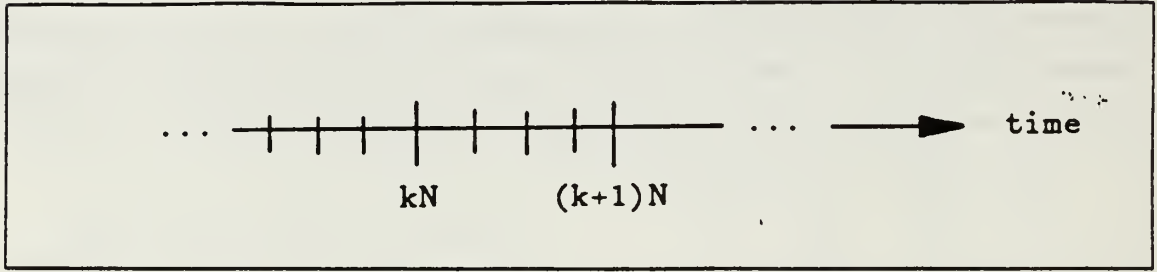

Figure 2.2 Time Line.

block, we reset the covariance matrix P(t). Although it can be reset at any time, for convenience we choose the end of each block, and P(t) now becomes

$$
P(t) = \begin{cases} \sigma_0^{-1}I & t = kN\text{-}1 \\ \text{Equation (2.17)} & \text{otherwise} \end{cases}
\qquad (2.21)
$$

Therefore, the beginning of each time block is treated as a new, initialized period.

It has been shown [Ref. 2: p. 20] that an external input u(t), sufficiently rich in frequency (n sinusoids), together with blocks I_k of sufficient length N provide a guarantee for an estimation $\theta \rightarrow \theta^*$ where θ^* represents actual parameters. The effect of various lengths of N will be investigated later.

If we apply the considerations above to the general case, we can see that the parameter estimates at the end of each time block are given by

$$
\begin{vmatrix} \phi^T((k+1)N-1) \\ \cdot \quad \cdot \\ \phi^T(kN) \\ \overline{---} \\ \sigma_0 I \end{vmatrix} \hat{\theta}_{(k+1)N} = \begin{vmatrix} y((k+1)N-1) \\ \cdot \\ y(kN) \\ \overline{---} \\ \sigma_0 \hat{\theta}_{kN} \end{vmatrix}
\qquad (2.22)
$$

This will be the foundation with which we build our recursive parallel algorithm.

Although the algorithm presented in this chapter assumes a single input, single ouput (SISO) plant, it can easily be extended to the multiple input-output (MIMO)

18

plant [Ref. 7: pp. 25-27].   Additionally, we assume our plant to be causal.
Furthermore, because the problem of order determination is necessarily complicated, it
will be assumed that the order of the plant is always known to the designer.

III. SYSTOLIC ARRAY IMPLEMENTATION

A. IMPLEMENTATION OF ALGORITHM

We now turn our attention to the solution of (2.22) using systolic arrays. In particular, (2.22) is suited for parallel computation. As described in [Ref. 2: pp. 23 - 25], the identification problem can be reconstructed into a set of linear equations as

$$R_k \, \theta_{(k+1)N} = \beta_{k1} \tag{3.1}$$

with R_k in upper triangular form. We see here that $\theta_{(k+1)N}$ can be computed by using two processors in cascade: one to compute R_k and β_{k1}, and the second to compute θ from (3.1). Note that implicit matrix inversion is not necessary since R_k is in upper triangular form.

As discussed in the previous chapter, we initialize the array at the beginning of each time block to $\sigma_0 I$ (left half) and $\sigma_0 \theta_{kN}$ (right half). This has the effect of initializing the R_k matrix in (3.1) to an upper triangular form. Then, at each discrete time n, an array of data $\varphi(n)$ and $y(n)$ will be passed to the systolic array. The task of the array is to "re-triangularize" the data at each clock pulse, so the matrix remains in the form prescribed by (3.1). It is then a simple matter to solve for $\theta_{(k+1)N}$. This technique is based on QR decomposition as the means to triangularize the data array.

The value of σ_0 relates to the confidence we have in the initial estimates. The larger the value of σ_0, the lower our confidence. Equations (2.18) - (2.22) illustrate the role that σ_0 plays.

B. GIVENS ROTATION

An attractive and easily implemented method of triangularization by QR decomposition is provided by the Givens rotations. The reason for this choice is the fact that the Givens rotation operates only on adjacent data, making it suitable for systolic array implementation. The object is to combine two adjacent rows in the matrix, forcing zeros in the appropriate positions so to obtain an upper triangular data matrix. Figure 3.1 shows an example of triangularization by successive Givens rotations.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11}' & a_{12}' & a_{13}' \\ 0 & a_{22}' & a_{23}' \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11}'' & a_{12}'' & a_{13}'' \\ 0 & a_{22}'' & a_{23}'' \\ 0 & a_{32}' & a_{33}' \end{bmatrix} \rightarrow \begin{bmatrix} a_{11}'' & a_{12}'' & a_{13}'' \\ 0 & a_{22}'' & a_{23}'' \\ 0 & 0 & a_{33}'' \end{bmatrix}$$

Figure 3.1   Example of Row Operations.

The underlying idea for this scheme is that any two rows of elements $a_{i,1} .. a_{i,n}$, $a_{i+1,1} .. a_{i+1,n}$ may be considered as a sequence of vectors (x,y) in $\mathcal{R}^2$ as in Figure 3.2.
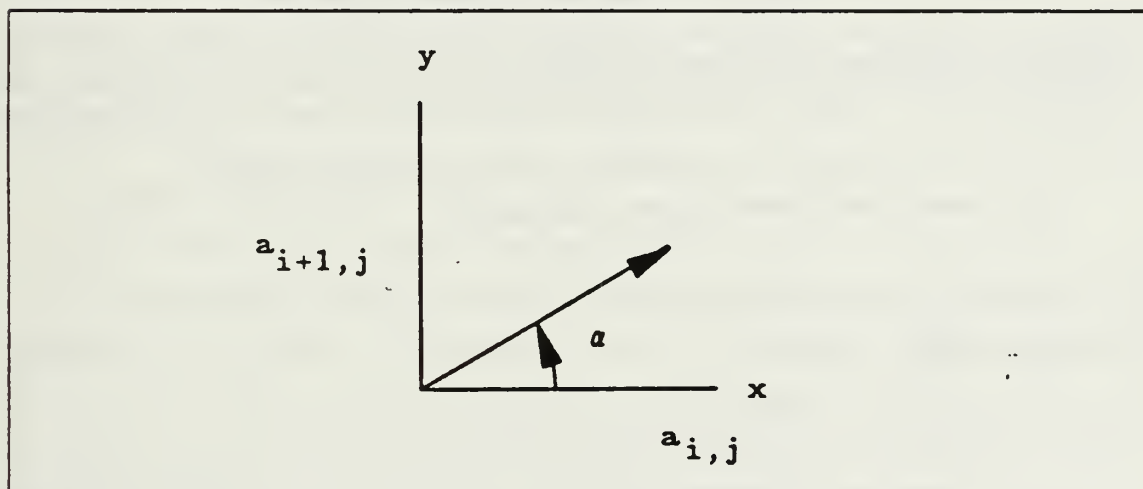
Figure 3.2   Vectors in $\mathcal{R}^2$.

If we rotate a vector $(a_{i,j}, a_{i+1,j})$ by an angle α so to make it parallel to the x axis, then the y value $(a_{i+1,j})$ becomes zero. This same rotation α is then applied to the remaining (x,y) vectors in the affected rows (in this case, rows i, i+1) to ensure algebraic equality. Next, the sequence of rotations is repeated for the remaining pairs of rows (i.e. rows i+1, i+2;   i+2, i+3; ...) in order to force zeros in the correct locations that leave an upper triangular matrix.

21

Recall that the data matrix is initialized to an upper triangular form ($\sigma_0 I$). Then, as we take samples from the signals in the plant, new values of $\varphi(t)$ and $y(t)$ are added to the matrix. By a sequence of Givens rotations, we can transform it into an upper triangular matrix. An example set of rotations is shown in Figure 3.3, where the

$$
\begin{bmatrix}
\phi_1\!\left(t\right) & \phi_2\!\left(t\right) & \phi_3\!\left(t\right) \\
a_{11} & a_{12} & a_{13} \\
0 & a_{22} & a_{23} \\
0 & 0 & a_{33}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
a_{11}' & a_{12}' & a_{13}' \\
0 & a_{22}' & a_{23}' \\
0 & 0 & a_{33}' \\
0 & 0 & 0
\end{bmatrix}
$$

Figure 3.3   Example of Givens Rotations.

operations performed by the systolic array at each clock pulse are illustrated. This is repeated until $t = kN$ (i.e. at the end of the time block), at which time the parameters θ are solved for, the matrix is reinitialized, and the process is repeated.

The basis of the Givens rotation is the matrix

$$
Q(p,q) =
\begin{vmatrix}
I_1 & 0 & 0 \\
0 & r(p,q) & 0 \\
0 & 0 & I_2
\end{vmatrix}
\tag{3.2}
$$

associated to each pair of indexes $p,q \in (1,n+1)$, with I_1 and I_2 being identity matrices of dimension $(q\text{-}2) \times (q\text{-}2)$ and $(n+1\text{-}q) \times (n+1\text{-}q)$ respectively, and r is a 2×2 matrix of the form

$$
r(p,q) =
\begin{vmatrix}
c(p,q) & s(p,q) \\
-s(p,q) & c(p,q)
\end{vmatrix}
\tag{3.3}
$$

The matrix Q has the property that it affects only two rows at a time, i.e. rows q and q-1. Application of the transformation $Q(p,q)$ to any matrix A of appropriate dimension yields

$$Q(p,q)A \;=\; \begin{vmatrix} x & x & x \\ x & 1 & x \\ x & 0 & x \\ x & x & x \end{vmatrix} \;\leftarrow\; q \qquad\qquad (3.4)$$

$$\uparrow$$
$$p$$

provided $c(p,q)$ and $s(p,q)$ in (3.3) are such that

$$c(p,q)\, a_{q-1,p} + s(p,q)\, a_{qp} = 1$$

$$(3.5)$$

$$c(p,q)\, a_{qp} - s(p,q)\, a_{q-1,p} = 0$$

In Figure 3.3, application of the Givens rotation $Q(3,4)Q(2,3)Q(1,2)$ to the left matrix results in the rotated matrix shown on the right. We see that to perform Givens rotations, we must be able to calculate addition, subtraction, multiplication, division, and squares. Next, we will see how to perform the rotations, by using add and shift operations only.

C.   CORDIC TECHNIQUE

The CORDIC (COordinate Rotation DIgital Computer) technique [Ref. 8: pp. 162-165] is particularly attractive in fixed point arithmetic for generating different types of trigonometric functions. The principle involved is to rotate a vector (x,y) through an angle α by a series of "properly quantized" angular steps. The single rotations of the vector (x, y) are computed at each step by a combination of simple add, subtract, and shift operations.

Consider again a vector in the plane as in Figure 3.2. The vector is represented by its x and y components. Rotating the vector through any angle δ leaves us with new rotated coordinates

$$x' = x \cos \delta \pm y \sin \delta$$
$$y' = y \cos \delta \mp x \sin \delta$$

$$(3.6)$$

where the top sign refers a clockwise rotation.

23

In the CORDIC technique, the rotation is performed in a sequence of angular steps ω_i, such that the sum of them approaches δ. The ω_i's can be positive or negative, so

$$\delta = \omega_0 \pm \omega_1 \pm \ldots \pm \omega_n \tag{3.7}$$

If we define $\gamma_i = \pm 1$, then we can express δ as

$$\delta = \sum_{i=0}^{n} \gamma_i \omega_i \tag{3.8}$$

The choice of the angles ω_i is determined on the basis of computational convenience. In particular, choose

$$\omega_i = \tan^{-1} (2^{-i}) \qquad i = 0, 1, \ldots \tag{3.9}$$

and consider a single rotation by an angle ω_i. By applying (3.6) and dividing by $\cos \omega_i$ we obtain

$$\frac{x'_i}{\cos \omega_i} = x_i \pm y_i \tan \omega_i = x_{i+1}$$
$$\frac{y'_i}{\cos \omega_i} = y_i \mp x_i \tan \omega_i = y_{i+1} \tag{3.10}$$

The factors $x'_i/\cos \omega_i$ and $y'_i/\cos \omega_i$ are the rotated components of the vector (x,y). Note that the new vector is not only rotated, but also scaled by a factor $1/\cos \omega_i$.

Now, combining (3.7) (or (3.8)), (3.9) and (3.10) we can define a recursion

$$x_{i+1} = x_i \pm y_i 2^{-i} = x_i + \gamma_i y_i 2^{-i} = k_i x'$$
$$y_{i+1} = y_i \mp x_i 2^{-i} = y_i - \gamma_i x_i 2^{-i} = k_i y' \tag{3.11}$$

for the rotation by an angle $\gamma_i \omega_i$. In binary arithmetic this can be implemented with simple shift and add operations. Figure 3.4 illustrates a typical system for the determination of γ_i.
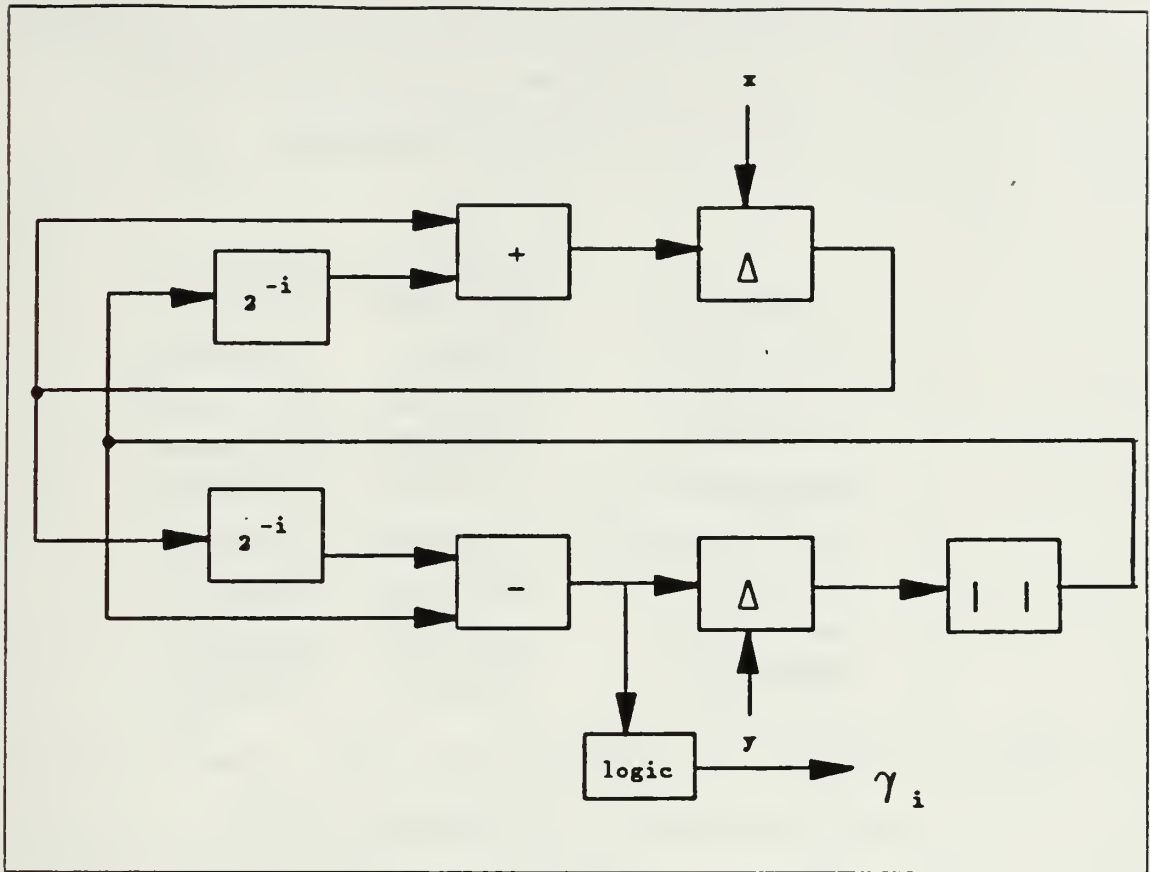
24

Figure 3.4   General Block Diagram of CORDIC Computation.

We can easily compute values of ω_i, $i = 0,1,2,...$ and the corresponding scaling factor k_i as given in Table 1.

For purposes of this research, we want to rotate an initial vector until its y value is equal to zero. The algebraic sum of the sequence of predetermined rotations will give us the angle δ that the vector was rotated through. These rotations can be encoded into a binary sequence γ that identifies the rotations performed. This sequence of rotations can then be passed to the remaining vectors in the affected rows. Figure 3.5 illustrates a series of rotations for an arbitrary vector which we desire to rotate by 30°. Note that in the figure, the desired angular value is approached quickly, because ω_i decreases by approximately half during each step. The γ sequence in the figure would be $(+1, -1, +1, -1, +1, +1, -1, +1, -1, +1, -1)$.

In an ideal case as just presented, the value of |y| decreases during each step. However, this may not always be the case. For example, consider the vector

TABLE 1

THE BINARY CORDIC CONSTANTS

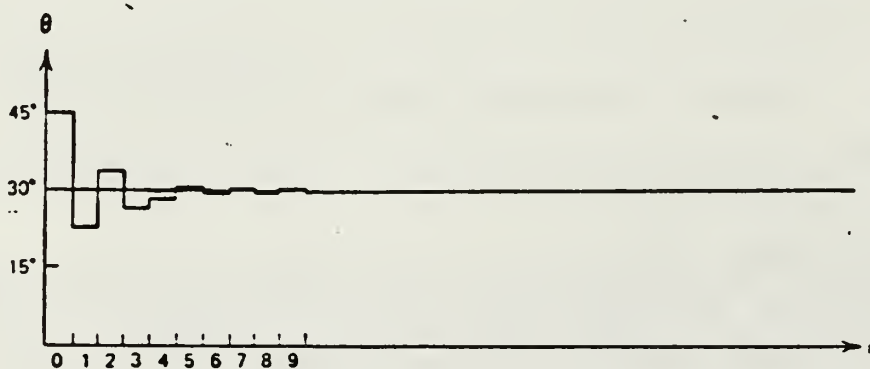| i	2^{-i}	ω_i (degrees)	k_i
0	1.0000000000	45.00000000	1.4142135
1	0.5000000000	26.56505124	1.581138826
2	0.2500000000	14.03624340	1.629800596
3	0.1250000000	7.12501632	1.642484060
4	0.0625000000	3.57633432	1.645688908
5	0.0312500000	1.78991064	1.646492240
6	0.0156250000	0.89517384	1.646639215
7	0.0078125000	0.44761428	1.646743467
8	0.0039062500	0.22381056	1.646756030

Figure 3.5 Set of CORDIC Rotations.

$(x,y) = (5,1)$ and $\alpha = 11.3°$. The first rotation in the CORDIC algorithm (see Table 1) would be $-45°$: this gives us a rotated vector $(x,y) = (4.2, -2.8)$ and $\alpha = -33.7°$. Note that the value of $|y|$ has actually *increased*. To make the algorithm more efficient, we modify the sequence γ to include the value zero. Whenever a rotation causes $|y|$ to increase, we do not perform it, and we set the corresponding γ_i to zero.

Then we continue on with the next rotation value and repeat the process. For the example just cited, the next value to be tried would be $26.6°$.

A simple algorithm to perform the CORDIC rotations can be seen in Figure 3.6.

INITIALIZE: if $x > 0$ then $x_0 = x$, $y_0 = y$

else $x_0 = -x$, $y_0 = -y$

if $y > 0$ then $\gamma_0 = +1$

else $\gamma_0 = -1$

for $i := 0$ to $N - 1$ do   (* number of iterations *)

if $y_i - \gamma_i 2^{-i} x_i > y_i$ then   (* |y| increases *)

$x_{i+1} := x_i$

$y_{i+1} := y_i$

$\gamma_{i+1} := 0$

else

$x_{i+1} := x_i + \gamma_i 2^{-i} y_i$

$y_{i+1} := y_i - \gamma_i 2^{-i} x_i$

if $y_i > 0$ then $\gamma_{i+1} := +1$

else $\gamma_{i+1} := -1$

end for;

end.

Figure 3.6   Algorithm for CORDIC Rotations.

D. SYSTOLIC ARRAYS

1. General

We now examine the parallel structure that will be used to solve the least squares algorithm described above. We desire a structure that accepts a sequence of regression vectors $\varphi(n)$ and signal $y(n)$ as input and then outputs the estimate for θ.

Specifically, we are interested in a high performance parallel strucuture that can be implemented directly as a hardware device in order to deliver maximum throughput. Systolic arrays represent a structure suitable for these characteristics.

The key to inexpensive implementation is simple and regular interconnections. Additionally, we want to allow the computations to proceed concurrently with the input, in order to maximize the throughput. This is known as *pipelining*. [Ref. 3: p. 257]
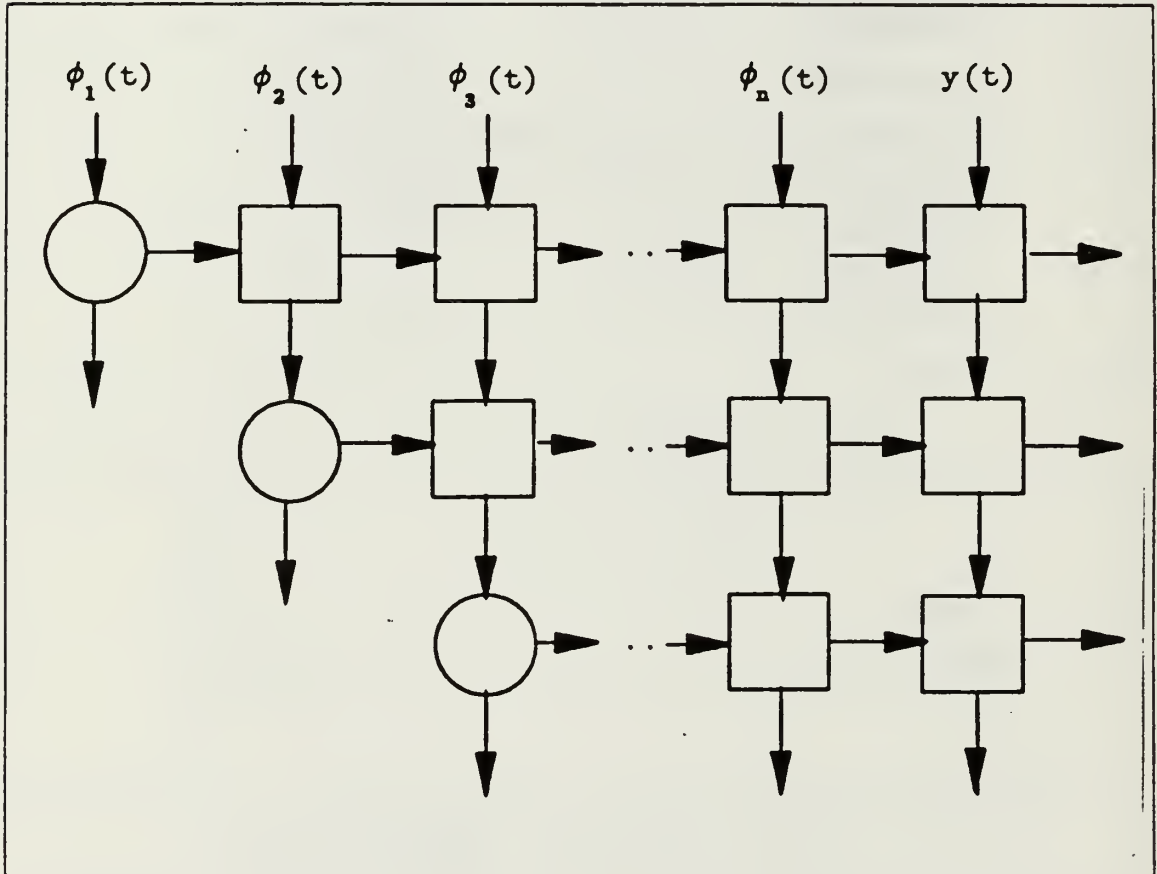

Figure 3.7   Systolic Array.

A systolic array meets these requirements. Figure 3.7 shows a typical system. As noted earlier, the array is simply a network of processors that are regularly connected. The data is continuously "pumped" through this structure, thereby minimizing overall execution time, since all the processors work in parallel.

It was shown that two processors would be necessary to solve (3.1). Previously used configurations have consisted of a triangular array (as in Figure 3.7) to compute

28

the upper triangular matrix, and a linear array to solve the system of equations. Figure 3.8 shows the typical design for the case where d (dimension) = 4. This thesis discusses an alternative configuration in which the linear section is replaced by a second triangular section identical to the first one. This new design is discussed later in this chapter. Both designs use a single clock signal to control operations. We will now review the structures of the triangular and linear sections before continuing on to discuss the alternative design.

2. Triangular Array

The triangular systolic array performs the rotations as described in the section on the CORDIC technique. The processors work simultaneously at each clock pulse. The data regression vector $\varphi(t)$ and output signal $y(t)$ is input to the top of the array, and rotations are calculated at each clock cycle.

The triangular array consists of two types of cells: edge (or boundary) cells and internal cells. The edge cells are represented by the circles in Figure 3.7 or 3.8; the boundary cells are the squares. Figure 3.9 defines the operations of these cells. The edge cells compute the rotation vector γ, which consists of a sequence of ± 1 or 0 as discussed previously. This vector γ is then passed to the internal cell on its right. The internal cell then rotates its (x,y) vector by the value specified by γ. These operations are performed down the length of the affected rows.

Each cell of the triangular array stores an element of the upper triangular matrix $R(n)$ from Equation (3.1), and it is initialized to zero for internal cells and to $\sigma_0 I$ for the edge cells. Then, each row of cells in the array is used to combine one row of the stored matrix with the data received from the above cells. As discussed previously, the array maintains its upper triangular form throughout the computations.

A delay of one clock cycle per cell is incurred when passing the rotation parameters along a row. Therefore, it is necessary to "skew" the input data as seen in Figure 3.8, so that the input data interacts properly with the previously stored triangular matrix. Because the cells are all operating simultaneously, the data in the system at any time t consists of values from (2n) different matrices. Figure 3.10 demonstrates this for a system with n = 3. In the figure, we can see that at time (t + 5), there are also values present from the five previous matrices (i.e. t + 4, t + 3,...,t). In order to get all the cells in the array to a similar time state, the array would have to be clocked an additional 2n-1 (five) cycles, feeding zeros as input where necessary. At the completion, all cells will be at the same time (t + 5).
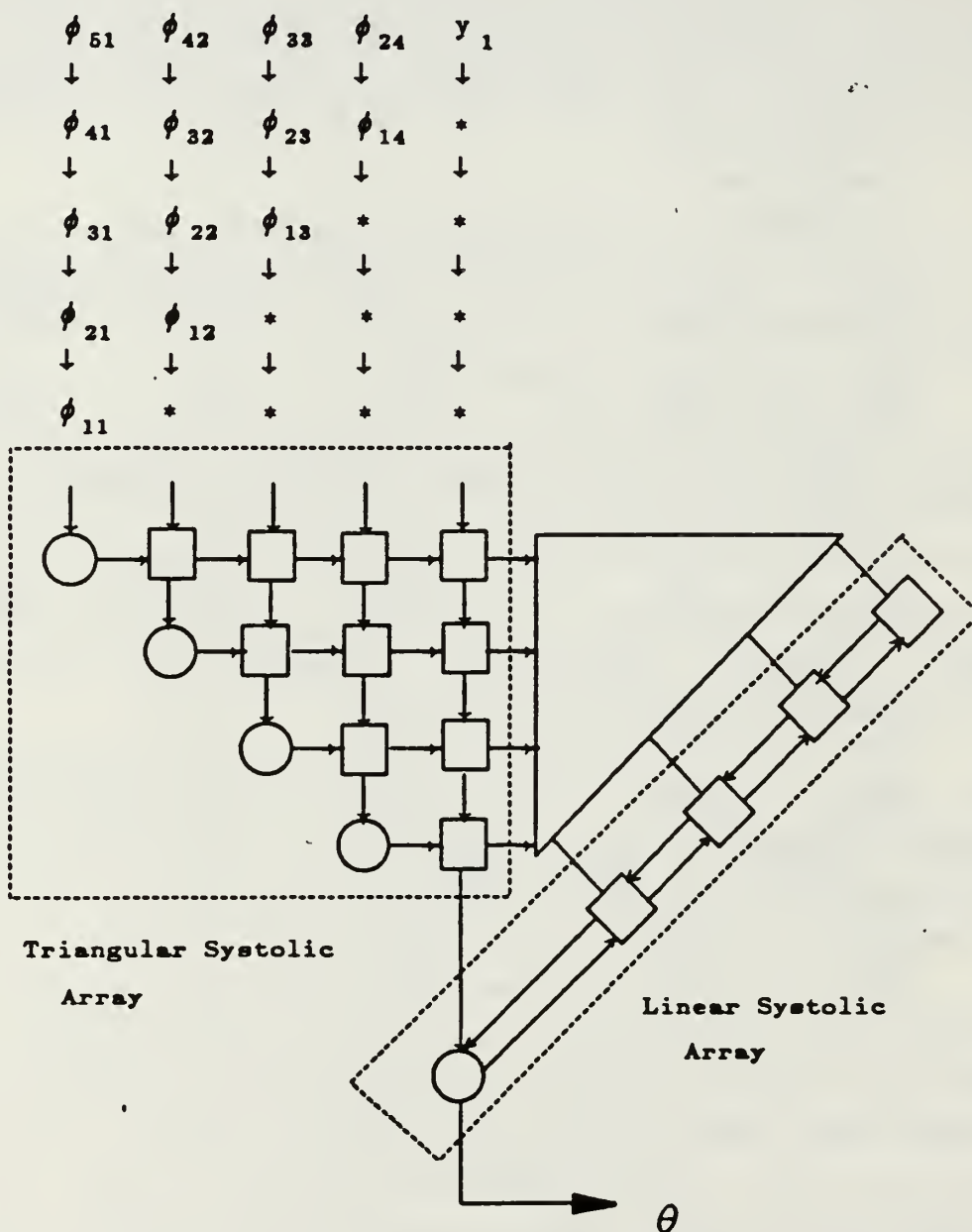
29

Figure 3.8   Systolic Array Implementation for Parameter Estimation: Old Design.

EDGE CELL

x' is computed recursively from

$$x_{i+1} = x_i + \gamma_i 2^{-i} y_i$$

$$\gamma_i = \begin{cases} +1 & \text{if } y_i > 0 \\ 0 & \text{if } y_i = 0 \text{ or the rotation} \\ & \text{would increase } |y| \\ -1 & \text{if } y_i < 0 \end{cases}$$

INTERNAL CELL

x' is computed recursively from

$$x_{i+1} = x_i + \gamma_i 2^{-i} y_i$$

y' is computed recursively from

$$y_{i+1} = y_i - \gamma_i 2^{-i} x_i$$

(over the sequence γ_i)

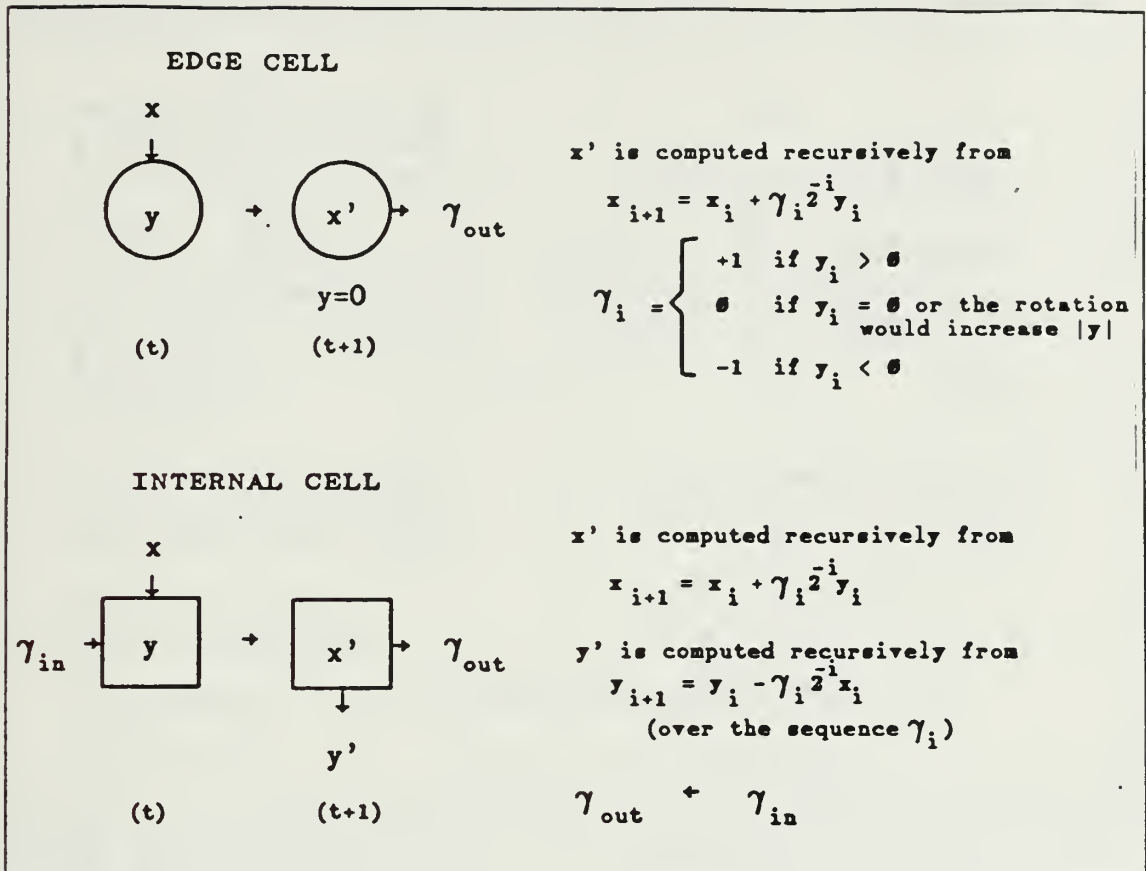$$\gamma_{out} \leftarrow \gamma_{in}$$

Figure 3.9   Definition of Cell Operations for Triangular Section.

Note that at the same time the triangularization process is being carried out, the column vector β_{k1} is also being computed by the right side column of internal cells using $y(n)$ as its input. At the end of the triangularization period (N), we are ready to solve for $\theta_{(k+1)N}$. The data in this triangular array is clocked out to the next array section that will compute the parameters.

3. Linear Array

The linear systolic array has been used in previous implementations to solve for the estimated parameters. The linear section consists of one boundary cell and (d-1) internal cells as seen in Figure 3.8. The operation of the cells as they compute the parameters are shown in Figure 3.11. Note that the cells are different from those in the triangular array, increasing to four the number of unique cells necessary in the combined system.
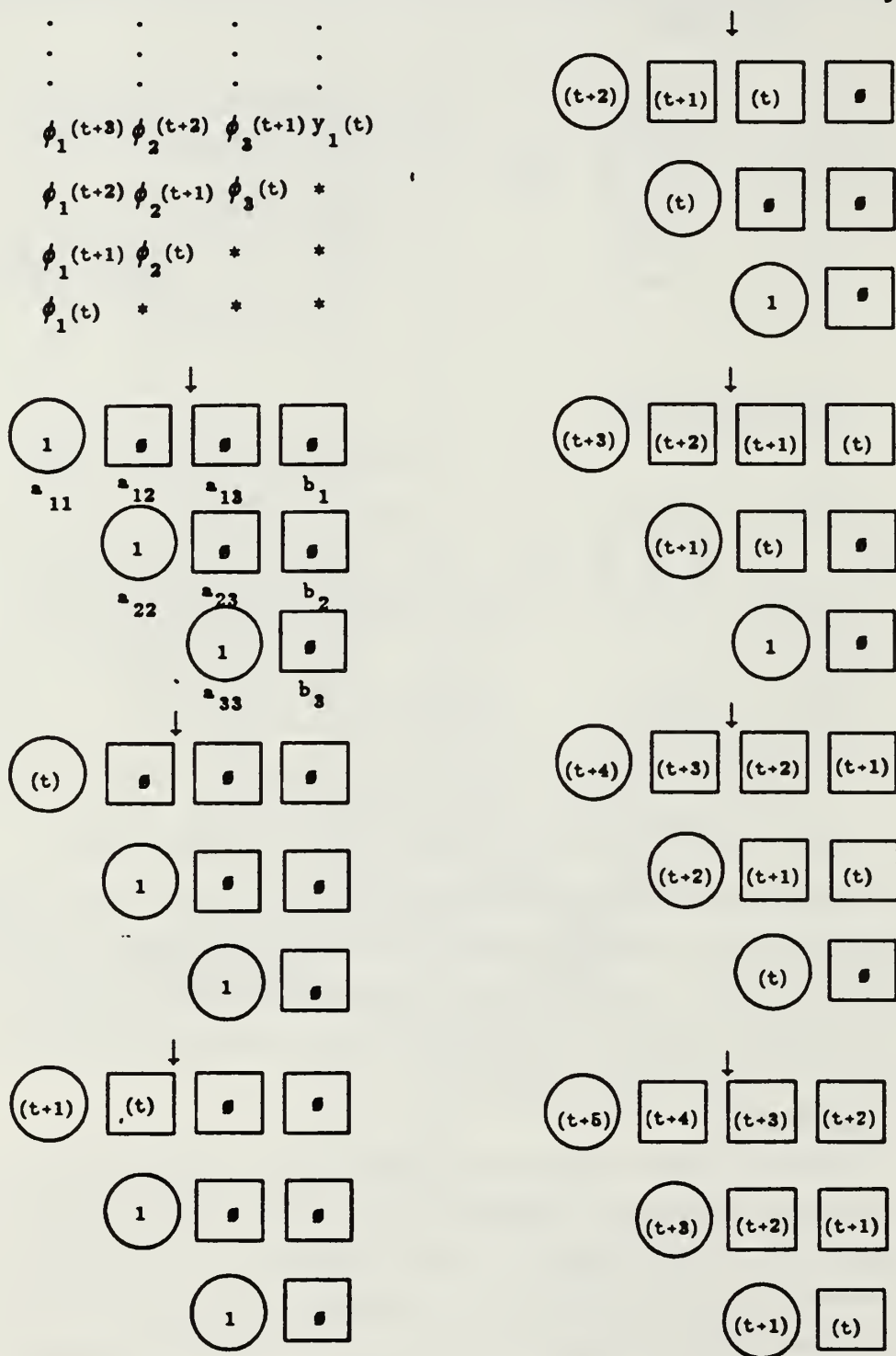
31

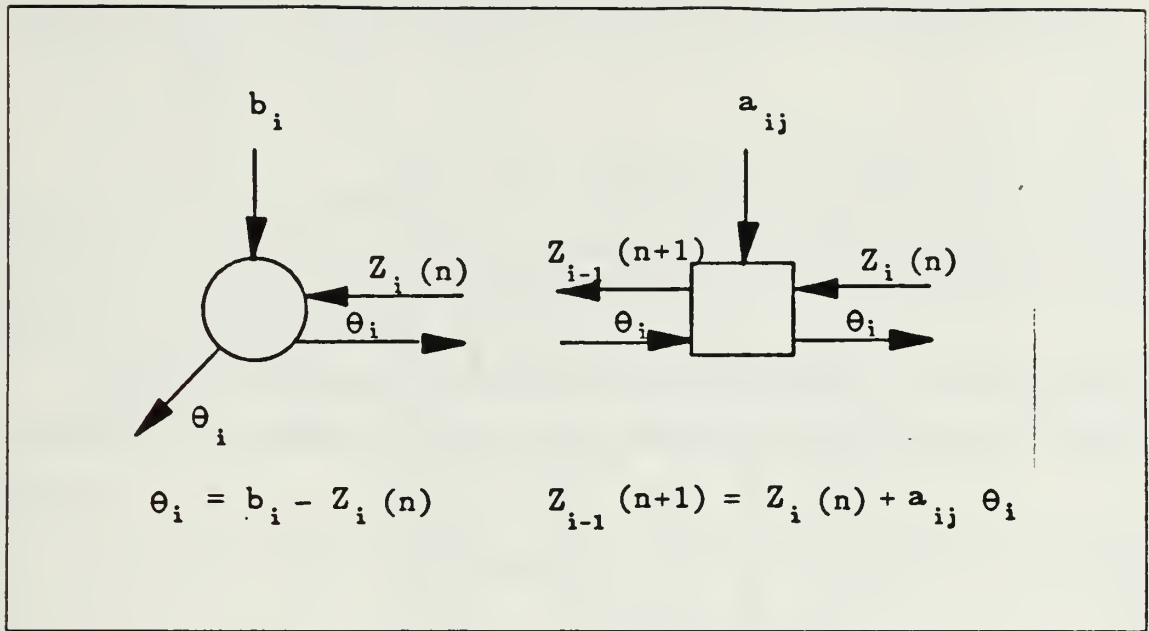Figure 3.10   Data Flow in Triangular Section.

32

$$\Theta_i = b_i - Z_i(n) \qquad Z_{i-1}(n+1) = Z_i(n) + a_{ij}\,\Theta_i$$

Figure 3.11   Definition of Cell Operations for Linear Section.

It is shown in [Ref. 2: p. 36] that the time required to solve for θ using the linear array is equal to 2d. At the end of that period, the parameters are used as initial values for the triangular array, and the triangular section again commences operation.

We now turn our attention to the replacement of the linear section by a second triangular system, and discuss the differences between the two designs.

4. The Use of a Second Triangular Array as the Solution Section

An alternative implementation can be obtained by solving for θ as shown in Figure 3.12. In this implementation, at the end of the triangularization period, the data is passed from the first triangular section to the second triangular section in a reversed fashion. The second array performs the same type of operations as the first; therefore the cells are identical.

The key to using another triangular section is that, by proper combinations of rows, we can force zeros into all elements of a given row but one, so that we can solve for each of the parameters. Also, the fact that orthogonal operations are used makes it more robust in the presence of numerical errors.

To see how this works, consider an arbitrary set of equations in upper triangular form as in Equation (3.12). Now, x_3 is simply solved as b_3/a_{33}. To solve for x_2, we can force a_{23} to zero by a linear combination of rows two and three. We know
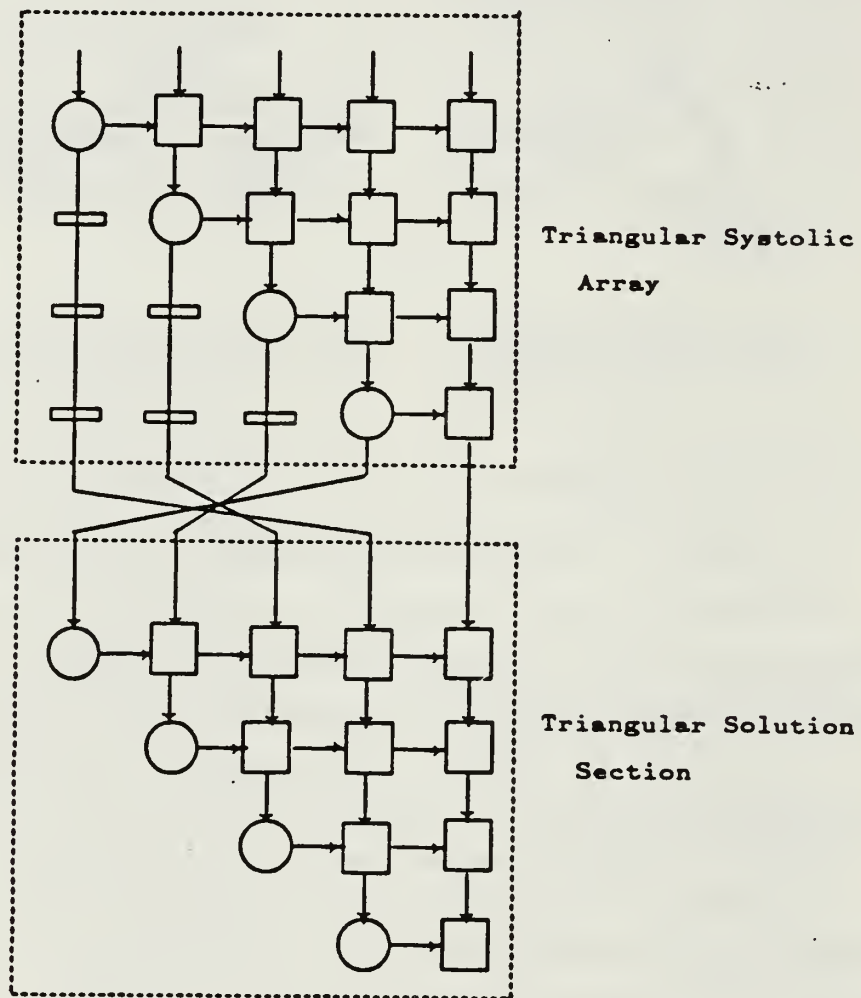
33

Figure 3.12   Systolic Array Implementation for Parameter Estimation: New Design.

$$
\begin{array}{rcl}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 & = & b_1 \\
0x_1 + a_{22}x_2 + a_{23}x_3 & = & b_2 \\
0x_1 + 0x_2 + a_{33}x_3 & = & b_3
\end{array}
\tag{3.12}
$$

this can be easily accomplished by Givens rotations. Then x_2 is found to be b_2/a_{22}. Similarly, by row operations on rows 1, 2, and 3 we can make $a_{12} = a_{13} = 0$ in row one. Again, solving for x_1 is a simple operation: b_1/a_{11}. These type of operations are exactly what the triangular array was designed to do.

Figure 3.13 illustrates these operations in matrix format. Note that the array is initialized to all zeros here, whereas the first triangular section is initialized to $\sigma_0\theta_{kN}$ and $\sigma_0 I$.
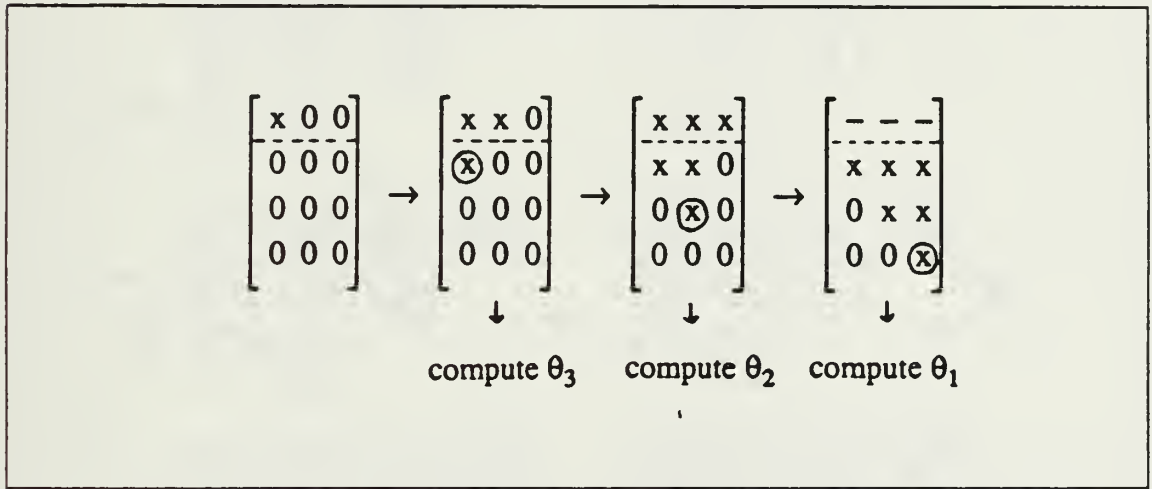

Figure 3.13   Solution of System of Equations.

To see how the system operates, recall the first triangular array at time N. Now we must feed the data down into the second triangular section in an appropriate manner so that we can solve for the parameters. The same delay (one clock cycle per cell) in propagation of data applies to this triangular section as it does in the first section. Hence, we must carefully choose when to sample the array in order to get the correct values with which to calculate the solution.

Figure 3.14 shows the data flow for a simple system where $d = 3$. The input data is skewed as it was in the triangular section. It can be seen that the values a_{33}, a_{22}, and a_{11} are available at times $N+1$, $N+4$, and $N+7$ respectively. Similarly,
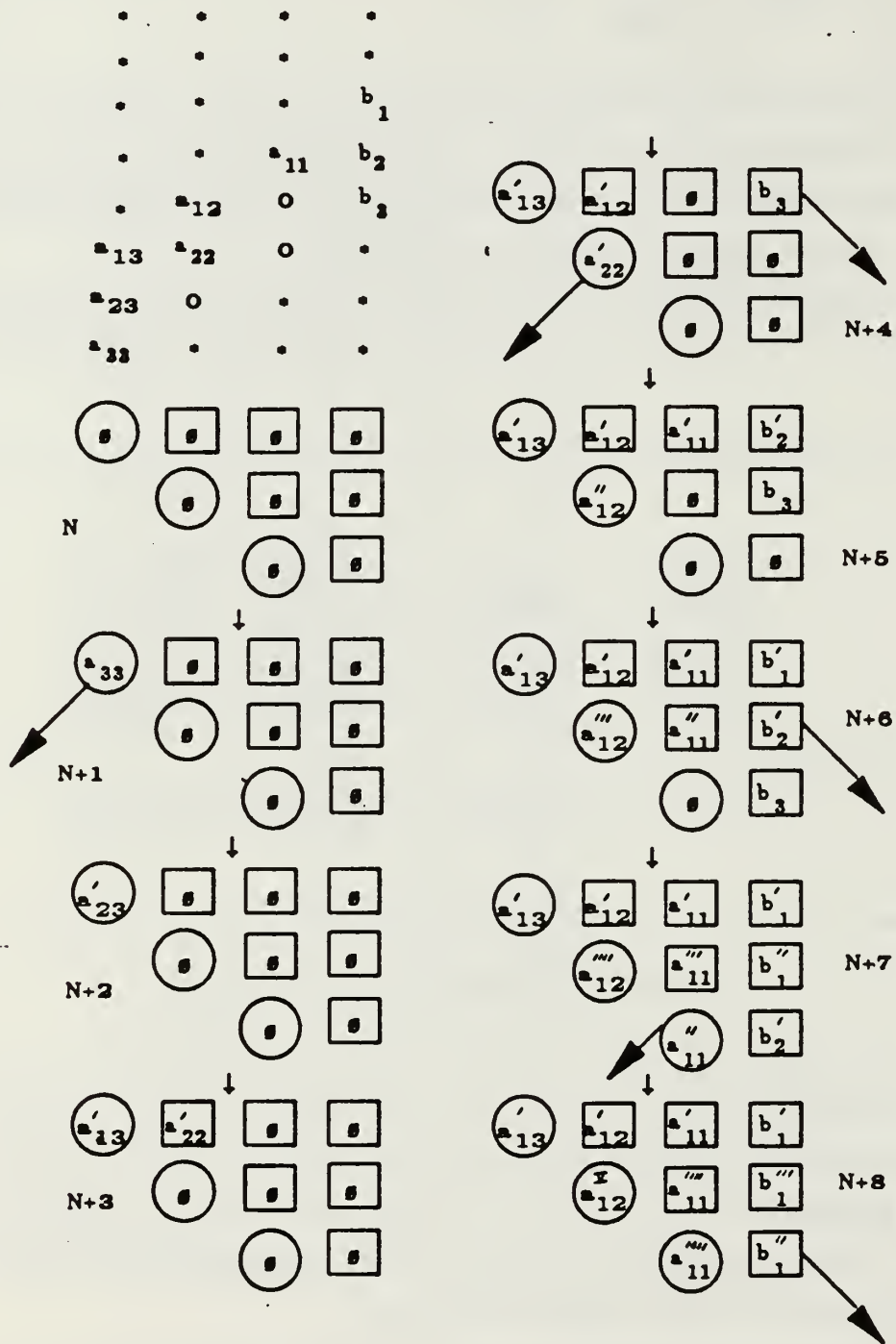
Figure 3.14   Data Flow Through Second Triangular Array.

values b_3, b_2, and b_1 are found at times $N+4$, $N+6$, and $N+8$. In general, for any size system n, the coefficients a_{ii} and outputs b_j are available as seen in Table 2. Note from the figure that the coefficients are "picked off" from the edge cells at the appropriate times, while the outputs are found in the rightmost set of internal cells.

TABLE 2

TIMES OF AVAILABILITY OF DATA

coefficient	time available
a_{nn}	$N + 1$
$a_{n-1,n-1}$	$N + 4$
$a_{n-2,n-2}$	$N + 7$
.	.
.	.
b_n	$N + (n + 1)$
b_{n-1}	$N + (n + 3)$
b_{n-2}	$N + (n + 5)$
.	.
.	.

This new design operates slower than the linear system seen in [Ref. 2: p. 36]. The time to solve for θ in this design is equal to the time until b_1 appears, which is equal to (3n - 1). This compares to 2n (or 2d) in previous implementations; hence, n-1 more clock cycles are required.

On the other hand, simplification is gained in the manufacturing process since the number of unique cells is reduced. Additionally, use of the CORDIC rotation technique allows us to use simpler processors. The tradeoffs to be considered are simplicity (cost) versus speed.

IV. SYSTOLIC ARRAY MODELING

A.   GENERAL

1. System Equations

We now study a particular "unknown" system and the performance of the systolic array in identifying its parameters. We will simulate a system both with and without noise present. Additionally, we compare the results obtained by using floating point versus fixed point arithmetic.

For purposes of the simulation, consider a plant with discrete time transfer function

$$H(z) = \frac{1.3z^2}{(z-0.5)^3} \qquad (4.1)$$

which corresponds to the linear difference equation

$$y(t) - 1.5y(t-1) + 0.75y(t-2) - 0.125y(t-3) = 1.3u(t-1) \qquad (4.2)$$

In particular, let the input sequence be

$$u(t) = \sin(3\pi t/10) + \sin(3\pi t/5) + \sin(3\pi t/2) + \sin(9\pi t/5) \qquad (4.3)$$

The dimension of the parameter vector is four, defined as $\theta = [-1.5, 0.75, -0.125, 1.3]$. In the parameter estimation problem, these values are assumed to be unknown. The input is "sufficiently rich" in frequency (minimum of n sinusoids) to excite all modes of the system [Ref. 1: p. 74]. Results of the simulations are discussed in the following sections.

2. Choice of Initial Values

For the recursive algorithm, recall that we need to initialize the systolic array with a parameter estimate $\sigma_0\theta(0)$ and $\sigma_0 I$. The value of σ_0, which is related to the confidence in the initial estimate as discussed in Chapter III, is chosen to be one for all simulations. If some information about $\theta(0)$ is available, it should be used when

38

determining appropriate initial conditions. In the absence of any prior knowledge of θ, we choose $\theta(0) = 0$.

3. Choice of Block Length

We have chosen four different block lengths during the simulation studies: $N = 3, 5, 10, 15$. It will be seen in the noiseless case that the parameters' exhibit the fastest rate of convergence when $N = 5$. However, when noise is present, there is a tradeoff: the system is more sensitive to disturbances when the block length is shorter. This is because a longer time block provides for more time averaging, thus attenuating the effects of noise. Therefore, with more samples available, disturbances have a lesser effect on the estimates. Hence, we must make an informed decision about what trait is most important in a specific application.

4. Fixed Point versus Floating Point

In the sections that follow, we also compare the performances obtained when using floating point processors or fixed point processors. Fixed point arithmetic operations are performed using finite registers and simple shift operations. Therefore, they are simpler to implement than floating point operations. Additionally, floating point operations in general require longer registers (exponent and mantissa) to represent numerical values, which might add complexity to the processor. Hence, the simplicity of fixed point arithmetic is desired.

The problem to be solved in the fixed point case is how to convert the input data values, which we normally expect to be floating point, into fixed point values. The answer, of course, is to appropriately scale all numbers so that they stay within the limits of the fixed registers. This task would be assigned to the Analog/Digital (A/D) converter, and the scale factor used would be in large part dependent on the range of values of the discrete plant samples.

B.   SYSTEMS WITHOUT NOISE

The system under consideration was first modeled in an ideal environment without noise to verify convergence of parameters. Figures 4.1 through 4.8 display the results of these simulations. In the figures, the estimated parameters (θ_1, θ_2, θ_3, θ_4) are plotted along the vertical axis, and the block number is plotted along the horizontal axis. Block number simply indicates the number of blocks that have been completed, where N identifies the block size. Specific results for the floating point and fixed point systems are discussed below.
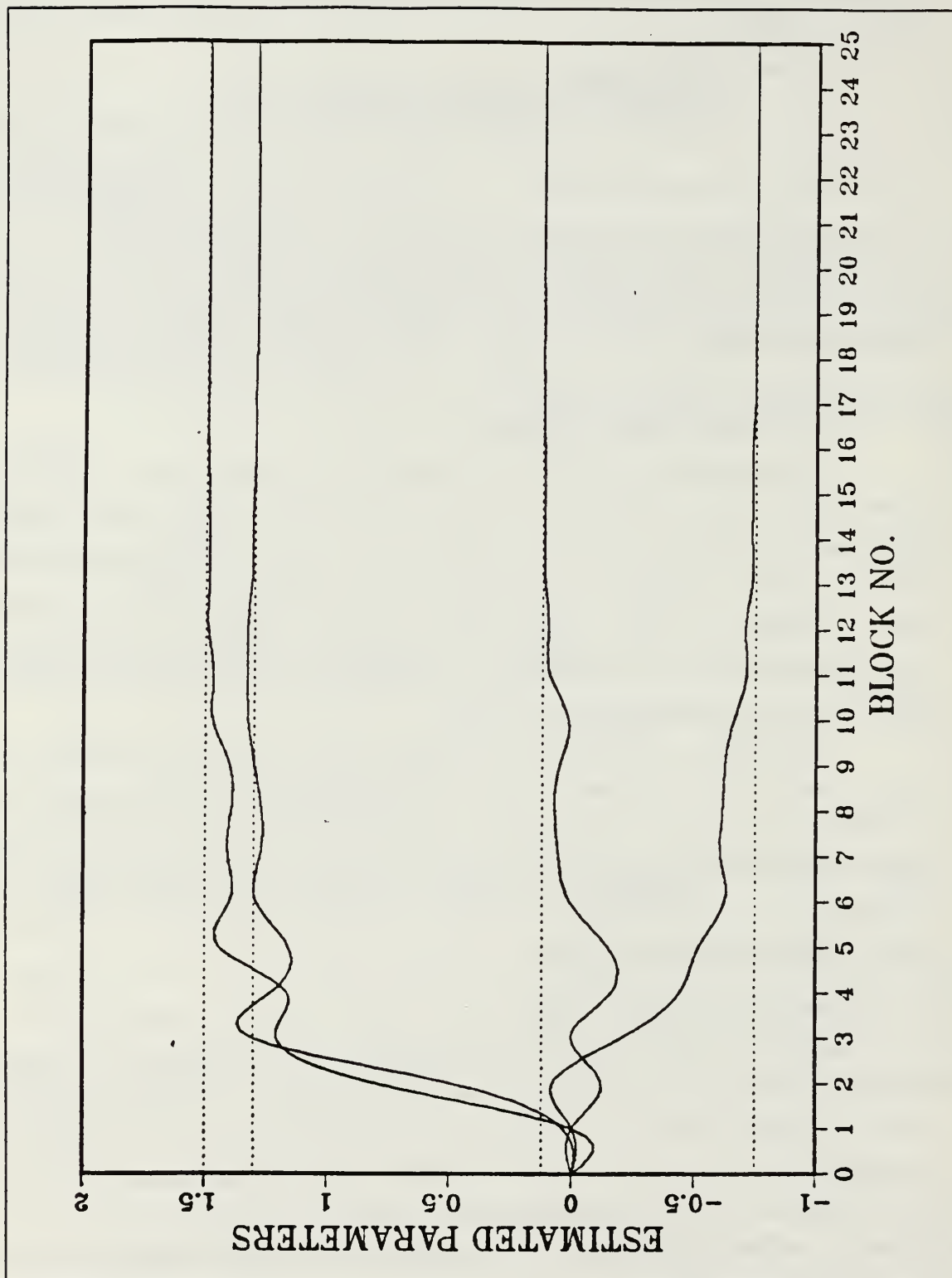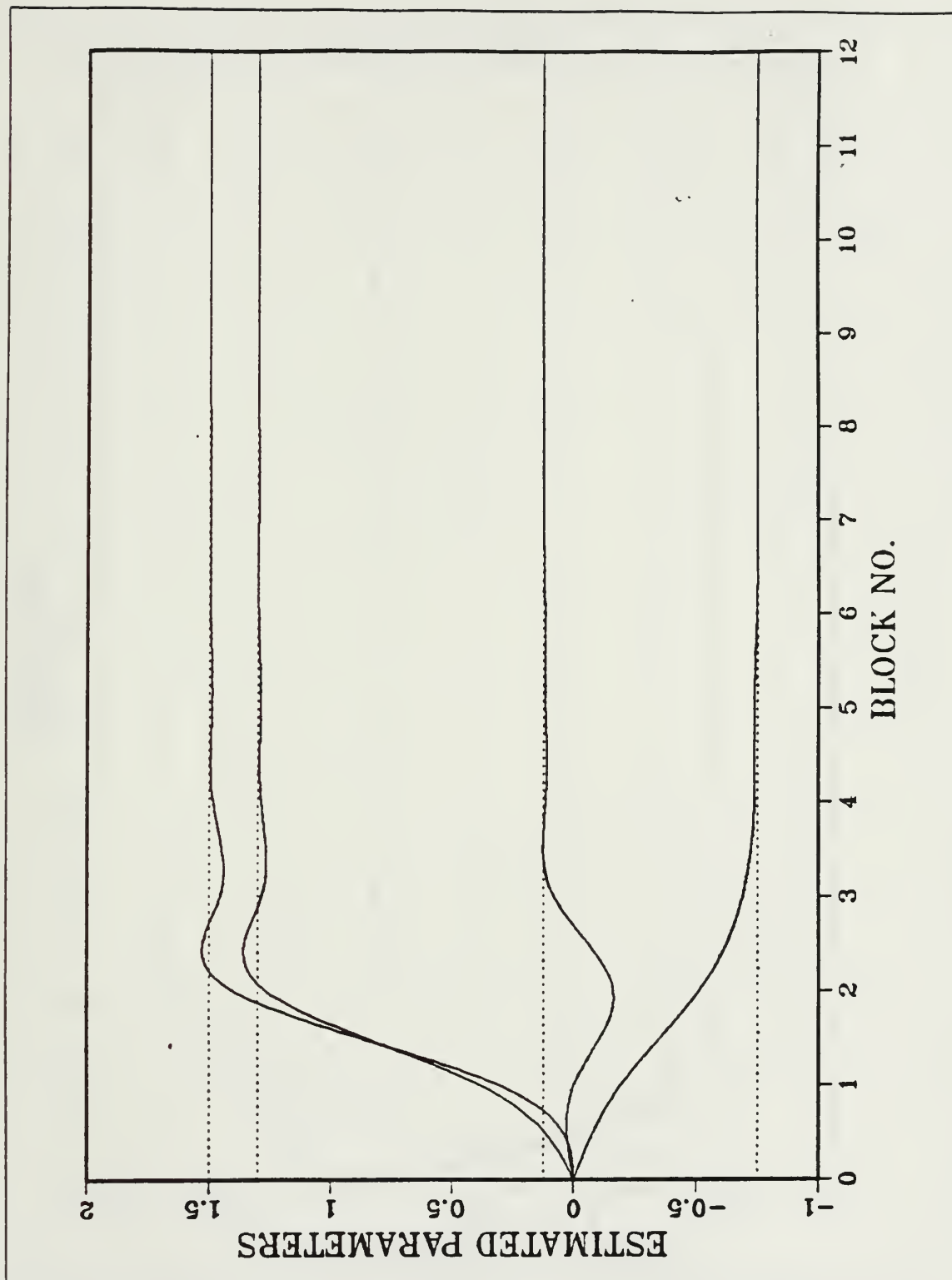
Figure 4.1 Floating Point, Without Noise, N = 3.
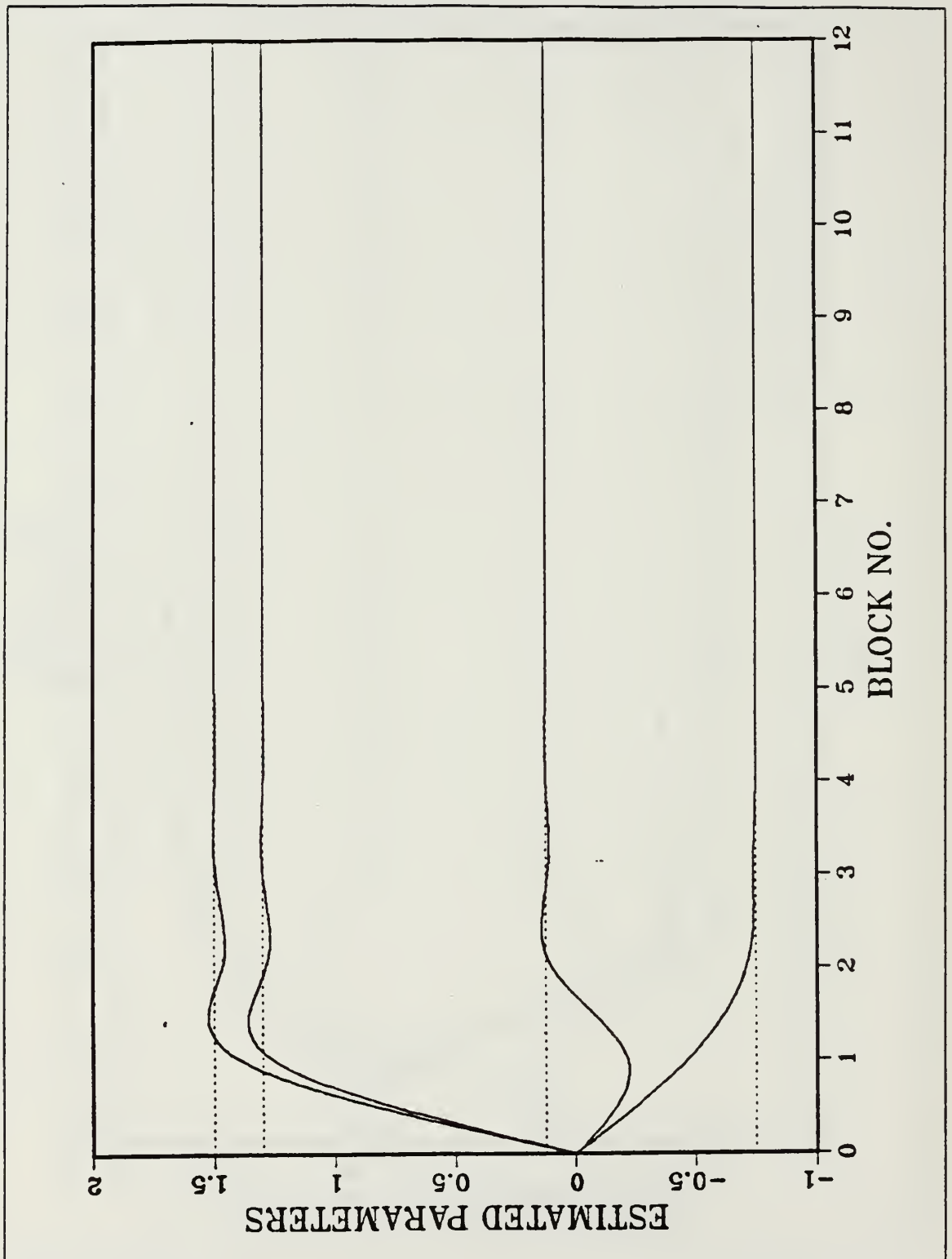
40

Figure 4.2   Floating Point, Without Noise, N = 5.

41

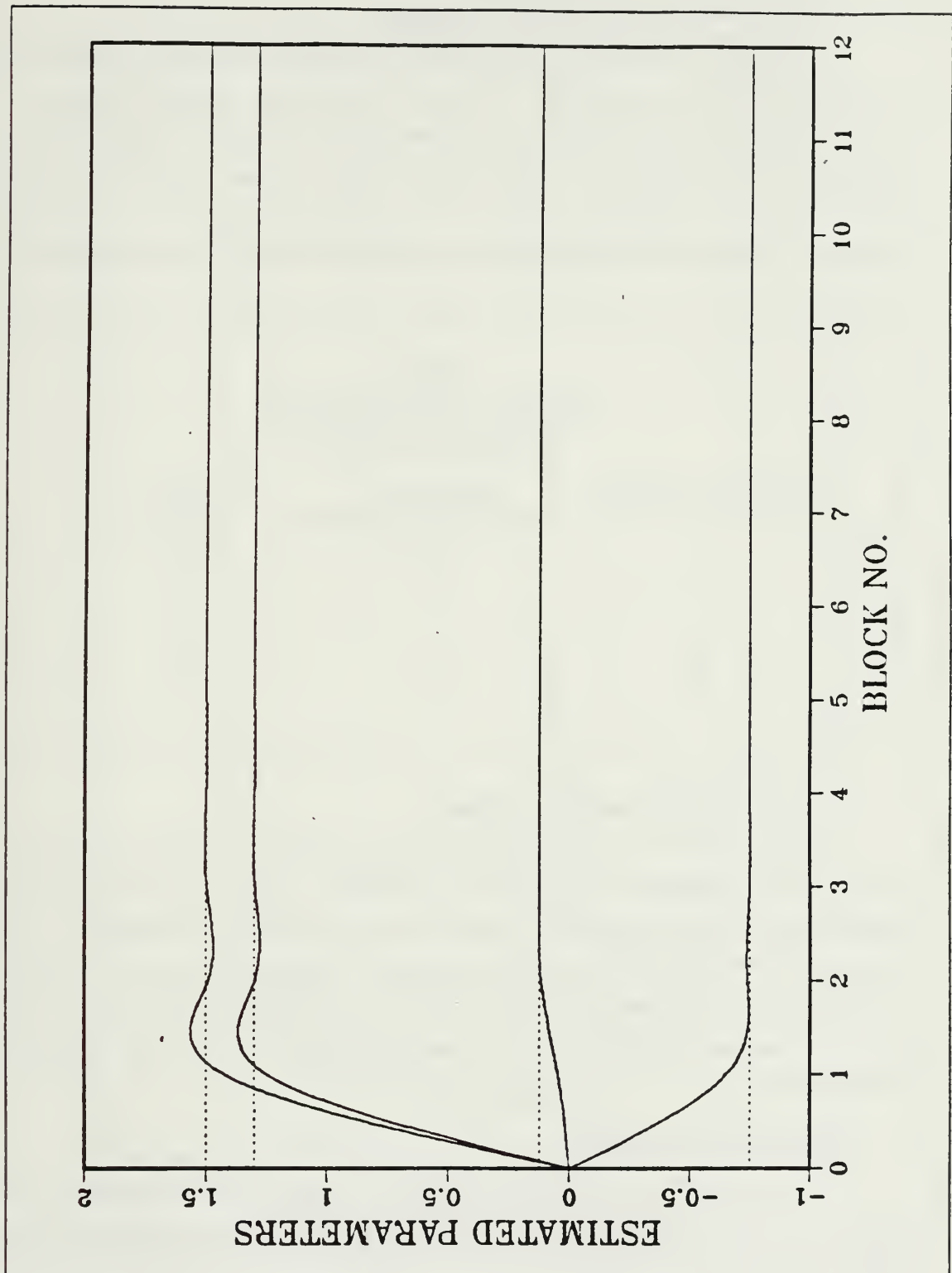Figure 4.3 Floating Point, Without Noise, N = 10.

42

Figure 4.4   Floating Point, Without Noise, N = 15.

43

1. Results Using Floating Point Arithmetic

Figures 4.1 through 4.4 show the floating point results for block lengths of $N = 3, 5, 10$, and 15 respectively. In order to evaluate and compare the relative rates of convergence, we estimate (by visual inspection) in each case where the parameters appear to have converged to their correct values. These values are tabulated in Table 3. It can be seen that with $N = 5$, the least number of clock cycles were required. Therefore, in this particular case, we should choose a block length of five.

TABLE 3
TIME TO CONVERGENCE
(NOISELESS CASE)

BLOCK LENGTH N	BLOCK NO. AT CONVERGENCE	TOTAL CYCLES
3	16	48
5	6	30
10	4	40
15	3.5	52

2. Results Using Fixed Point Arithmetic

The results for the fixed point system are shown in Figures 4.5 through 4.8. Note that the parameters converge just as rapidly as they did in the floating point implementation. (The simulations were run on an IBM 3033 mainframe, in which there are 31 bits available for an integer number.) This indicates that the systolic array with fixed point processors performs as well as the floating point system. This is a distinct advantage when we consider the simplicity of fixed point processors as discussed previously.

C.   SYSTEM WITH NOISE

The system was next modeled with noise present. The noise term $v(t)$ is a sequence of independent random variables identically distributed with zero mean (as white noise). We use a variance of 0.5 for these noise random variables. The noise is added to both the incoming signal $u(t)$ and measured output $y(t)$, as would be expected in a real system. Figure 4.9 illustrates these signals both with and without noise.
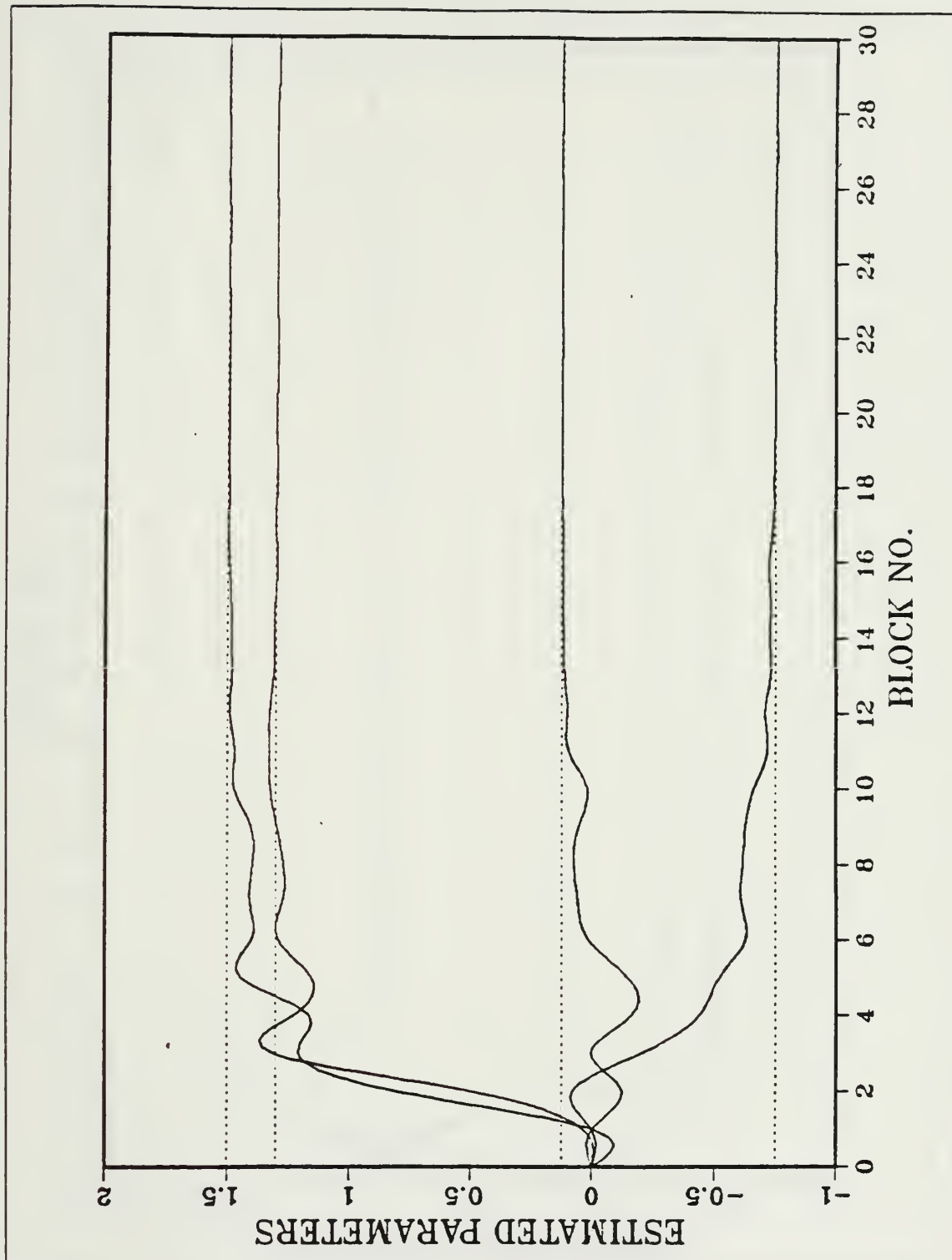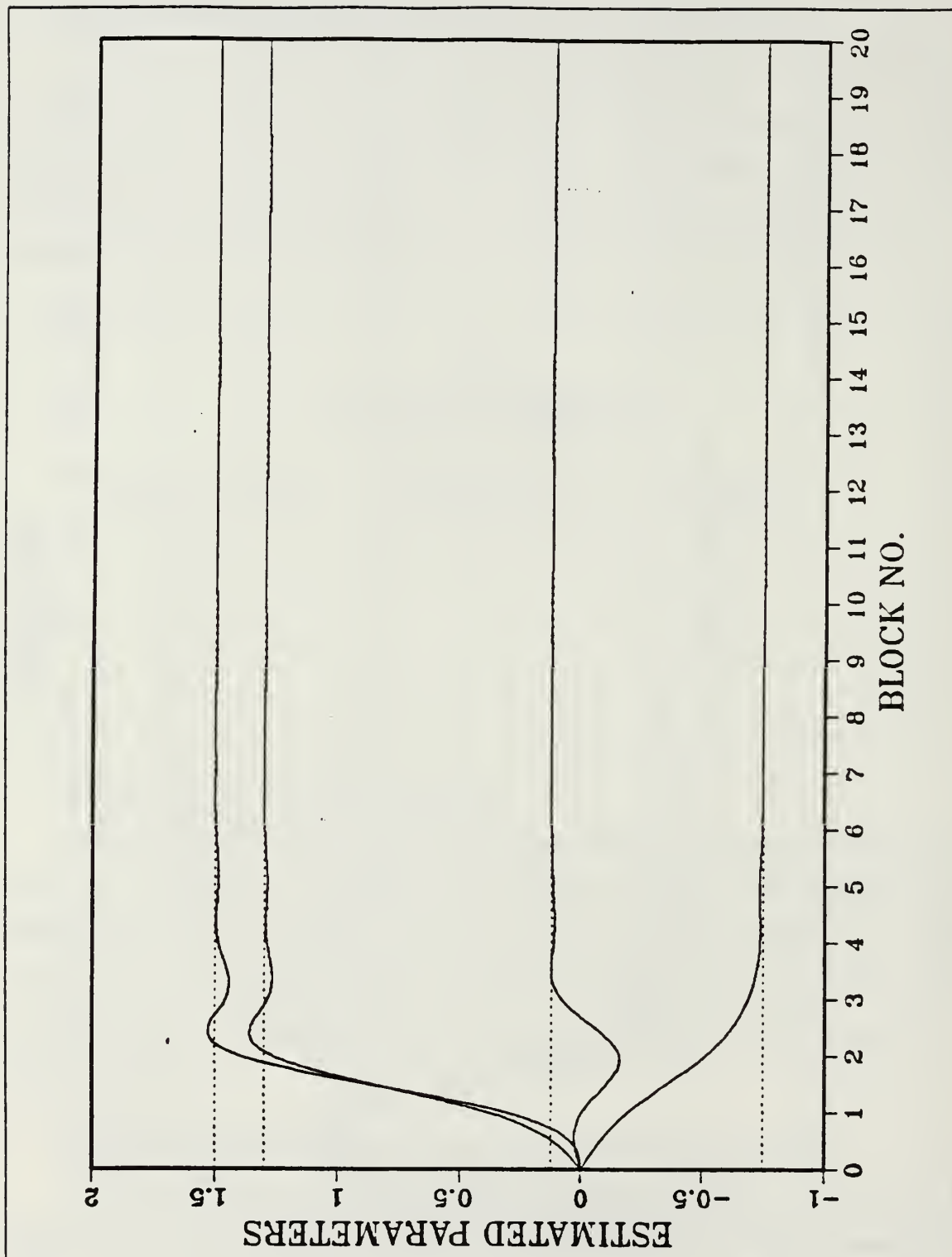
Figure 4.5   Fixed Point, Without Noise, N = 3.

45

Figure 4.6   Fixed Point, Without Noise, N = 5.
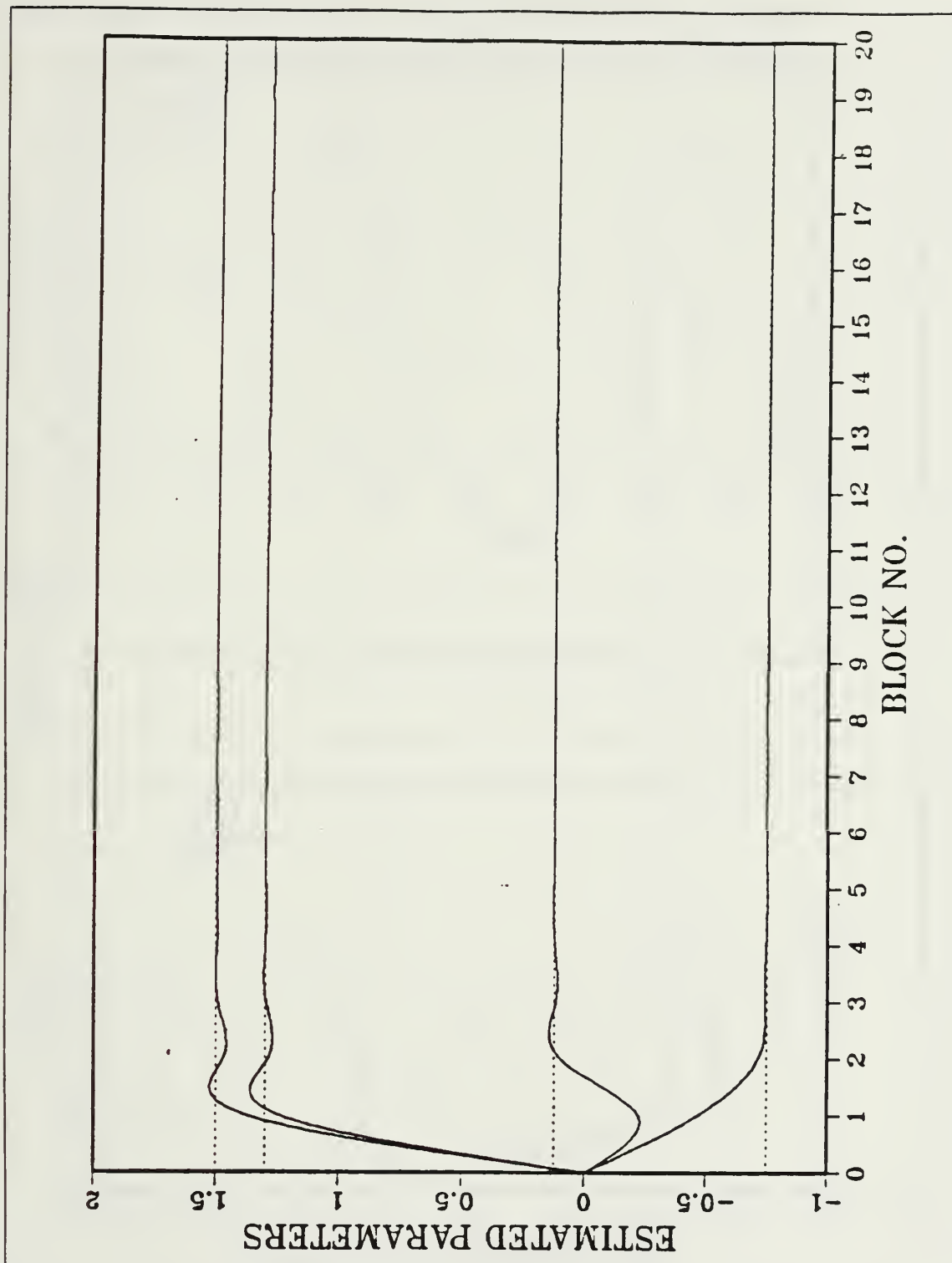
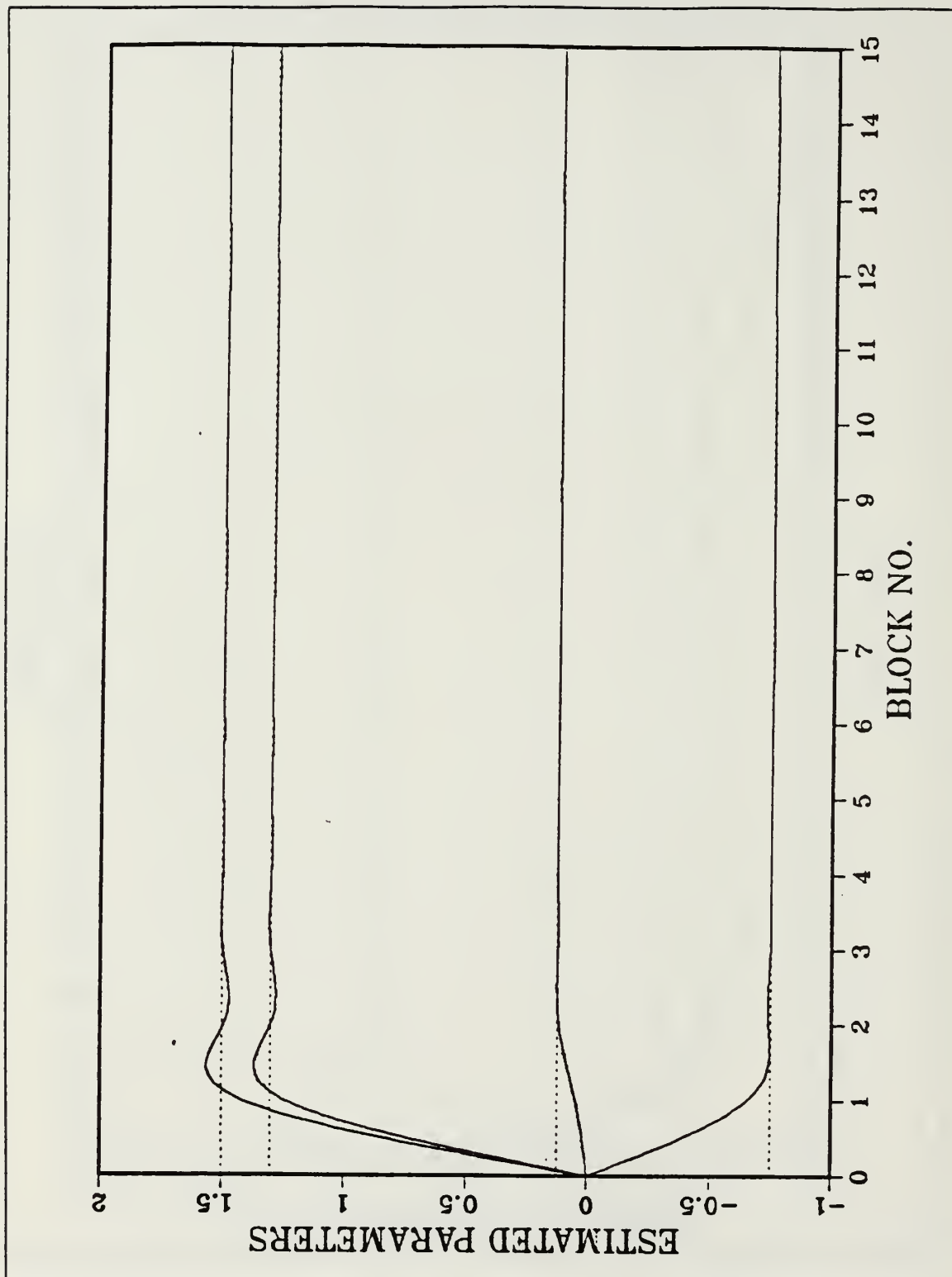Figure 4.7   Fixed Point, Without Noise, $N = 10$.
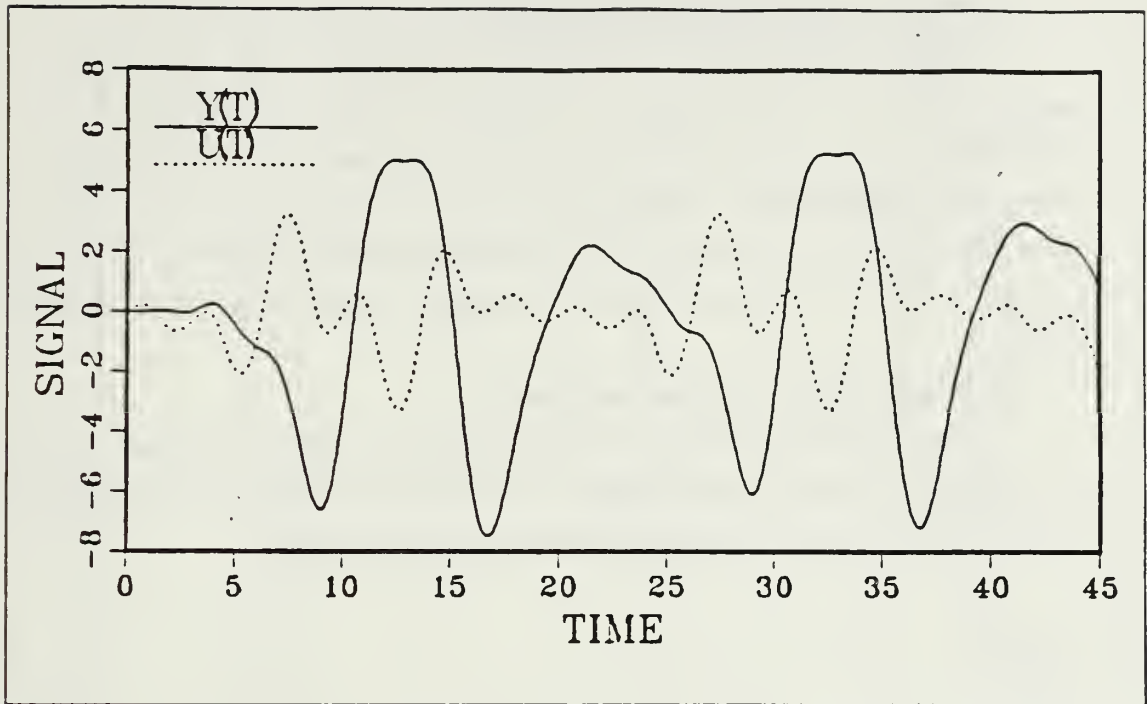
47

Figure 4.8   Fixed Point, Without Noise, N = 15.

48

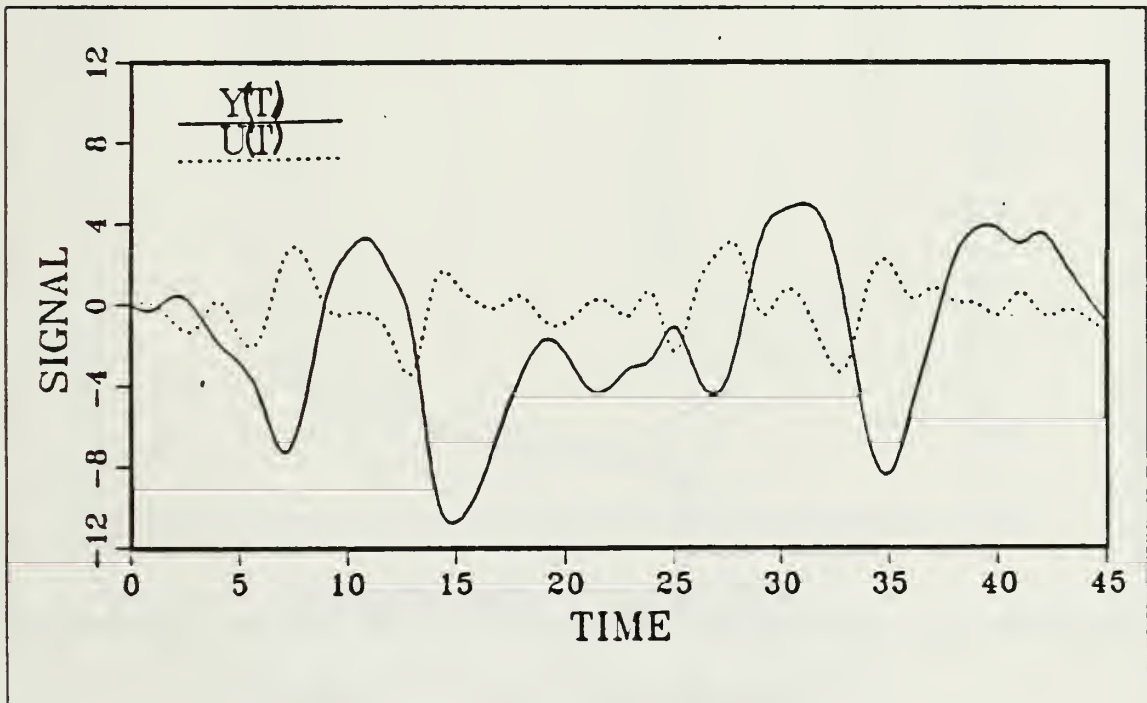Figure 4.9a   Input and Output Signals Without Noise Present.

Figure 4.9b   Input and Output Signals With Noise Present.

49

1. Results Using Floating Point Arithmetic

Figures 4.10 through 4.13 display the results for the four different block lengths. With $N = 3$, noise causes the parameters to vary considerably. When $N = 5$ or 10, disturbances are less likely to affect the parameters. When block length is equal to 10, the parameters have converged reasonably well within three blocks (or a total of 30 cycles). For $N = 15$, we see that the parameters are least affected by the noise, as expected. Here we find the relative convergence time is about three blocks, or 45 cycles.

2. Results Using Fixed Point Arithmetic

The results for fixed point implementation are shown in Figures 4.14 through 4.17. Again we note that they exhibit the same performance as in the floating point cases. Effects of block length are the same as noted previously.

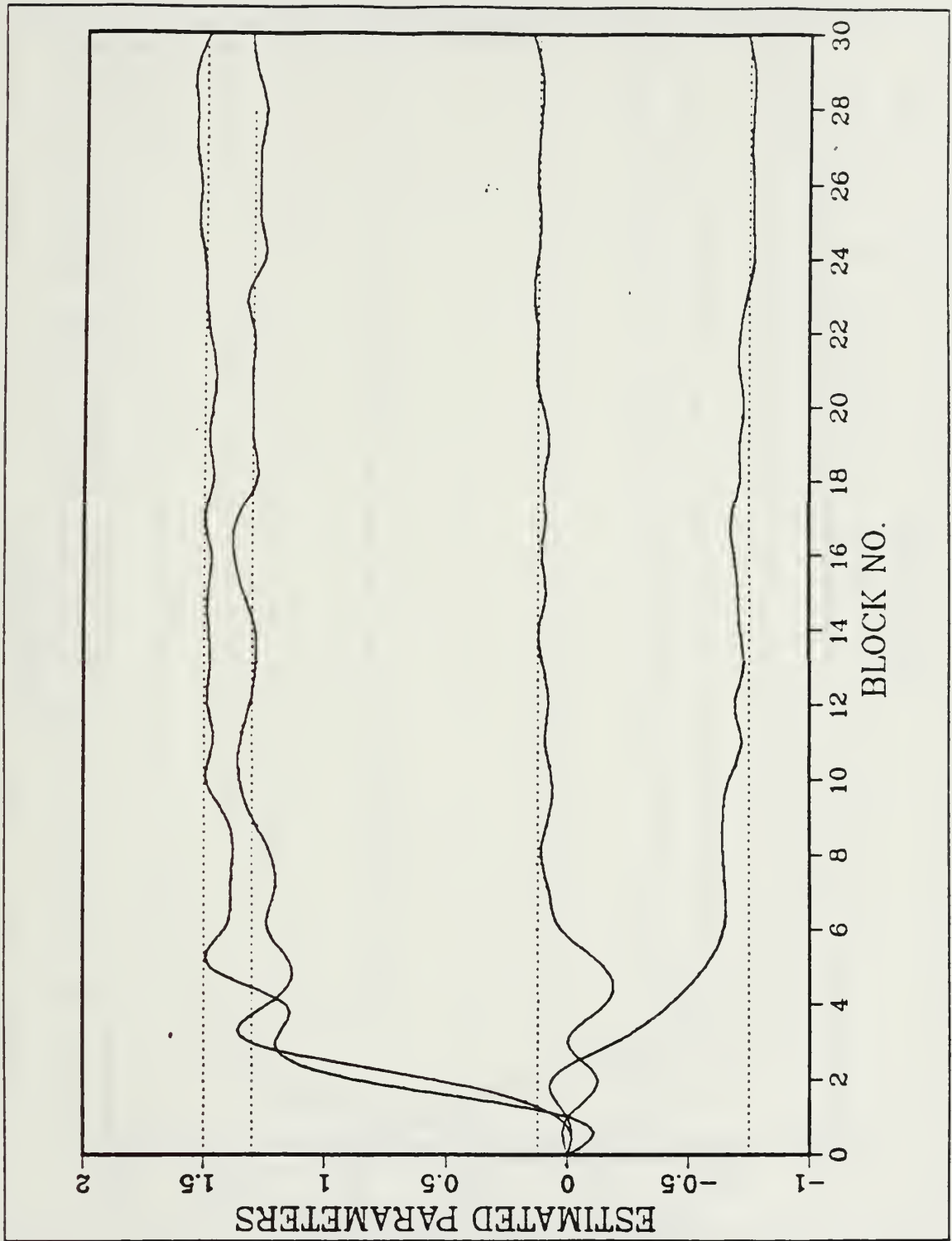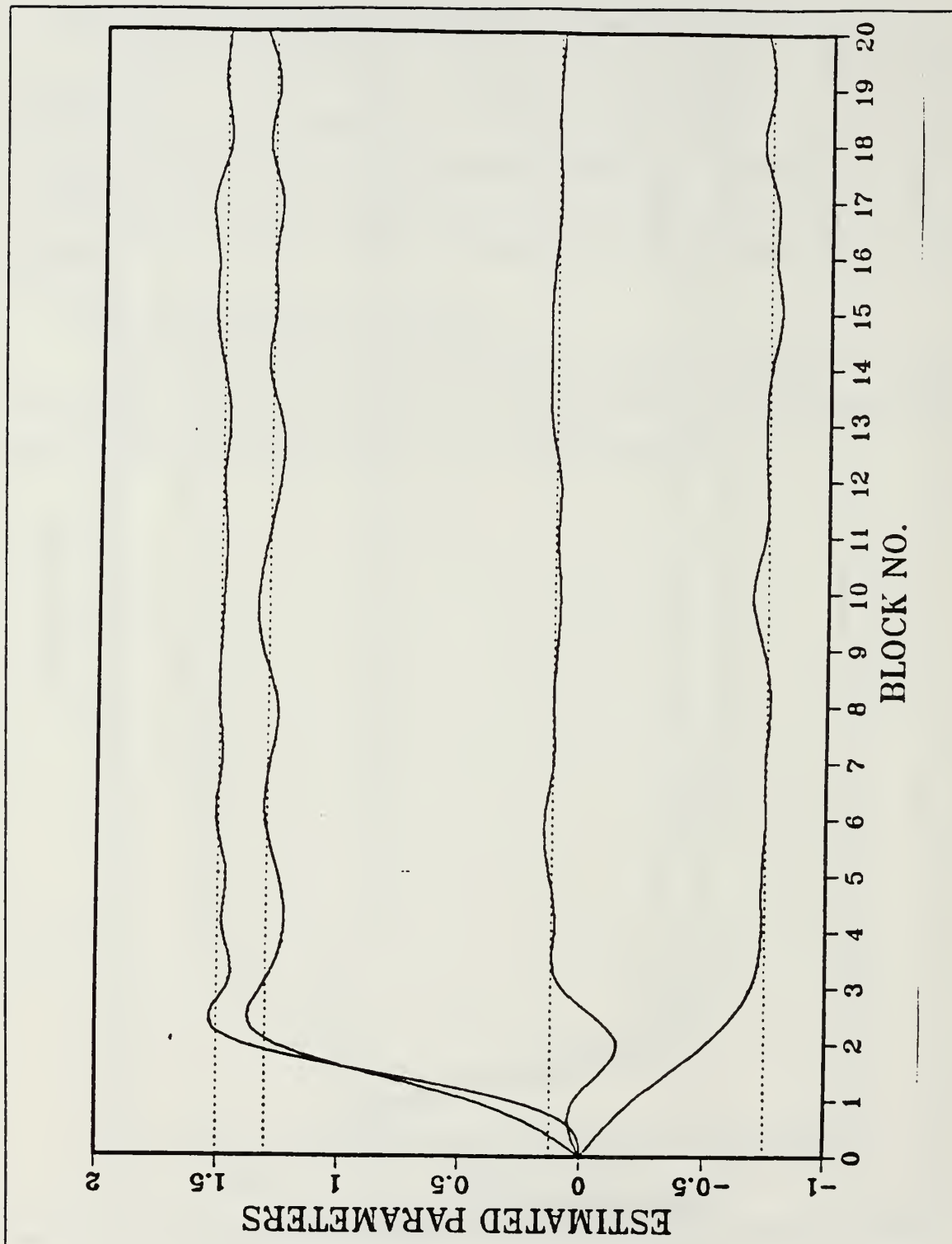Figure 4.10   Floating Point, With Noise, N = 3.

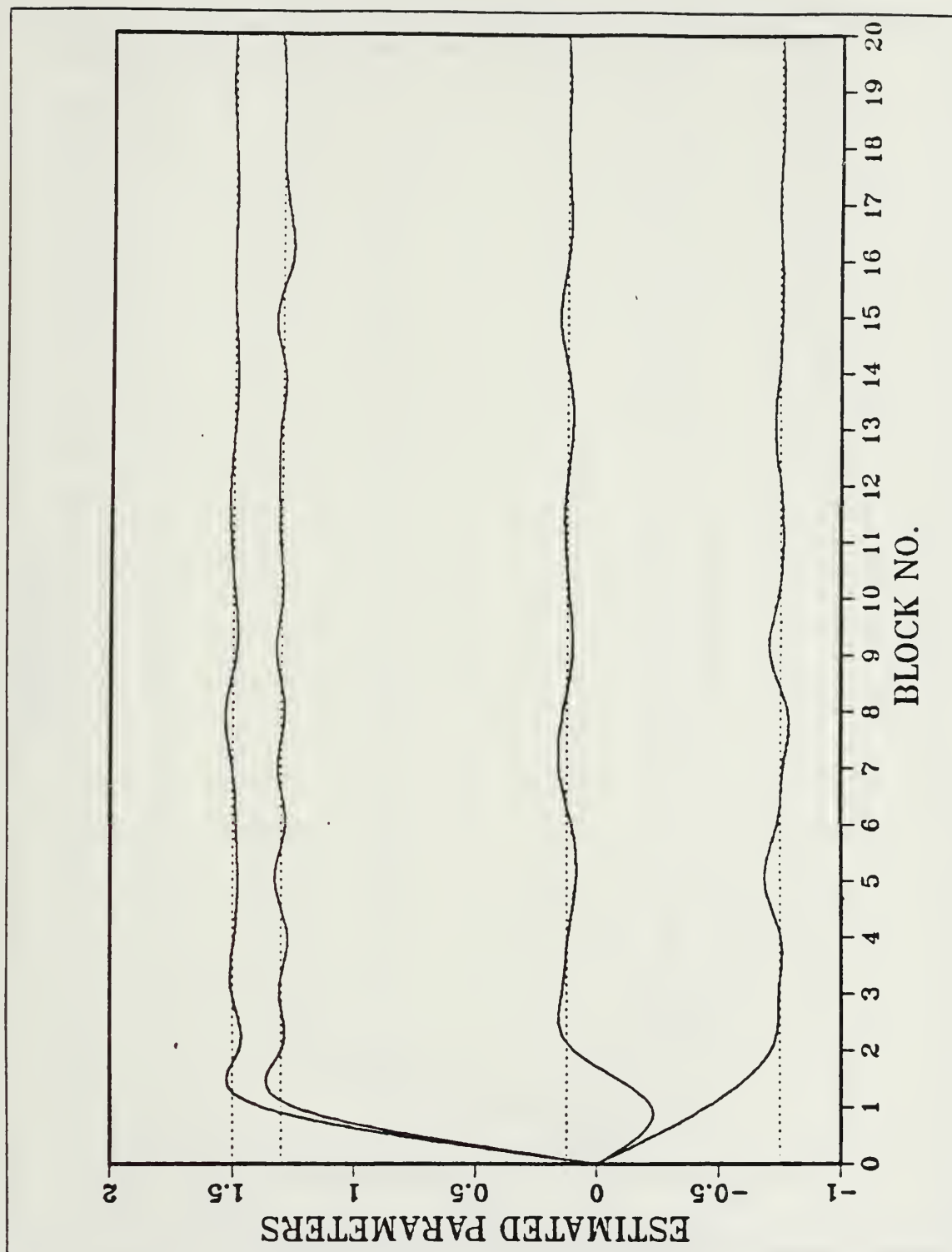Figure 4.11   Floating Point, With Noise, N = 5.

52

Figure 4.12   Floating Point, With Noise, N = 10.

53

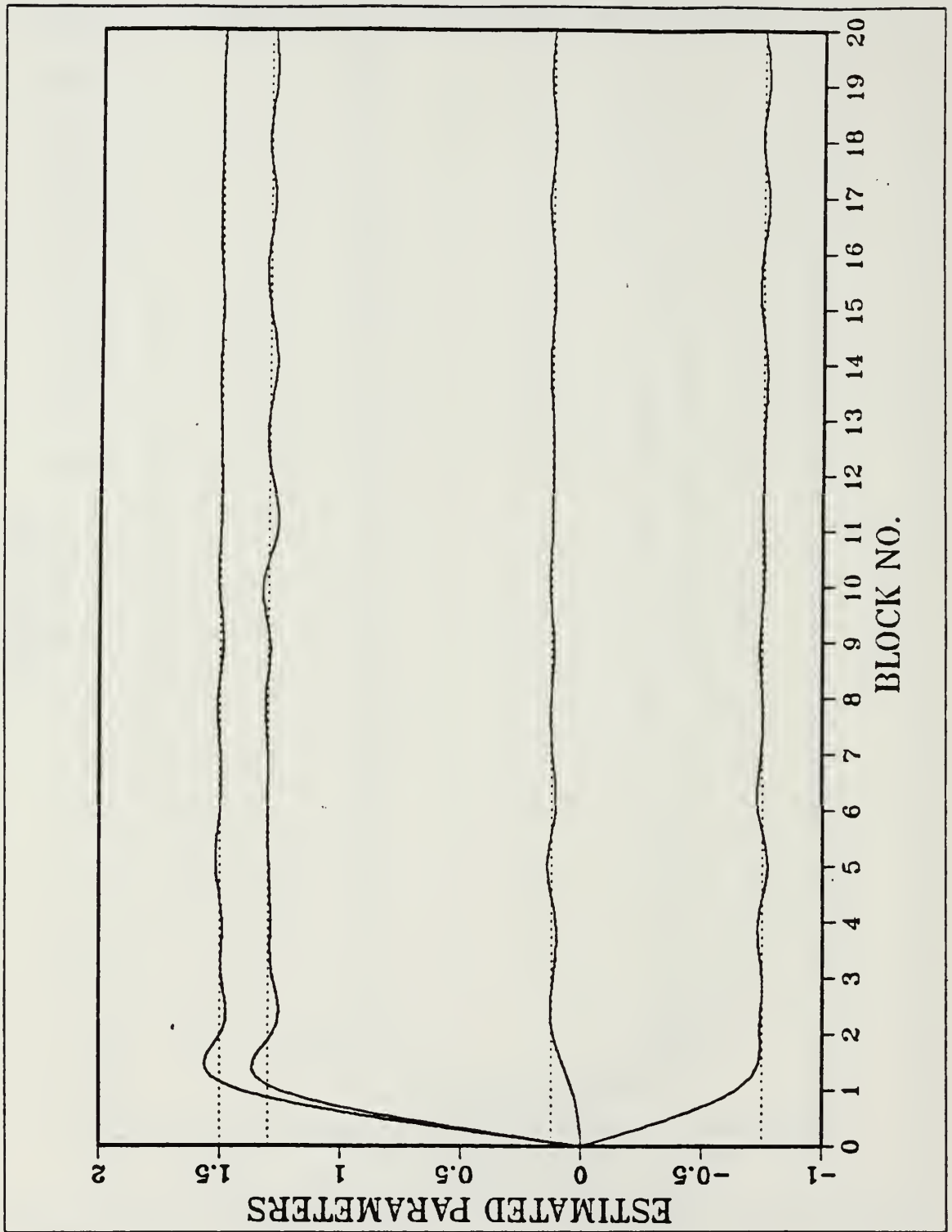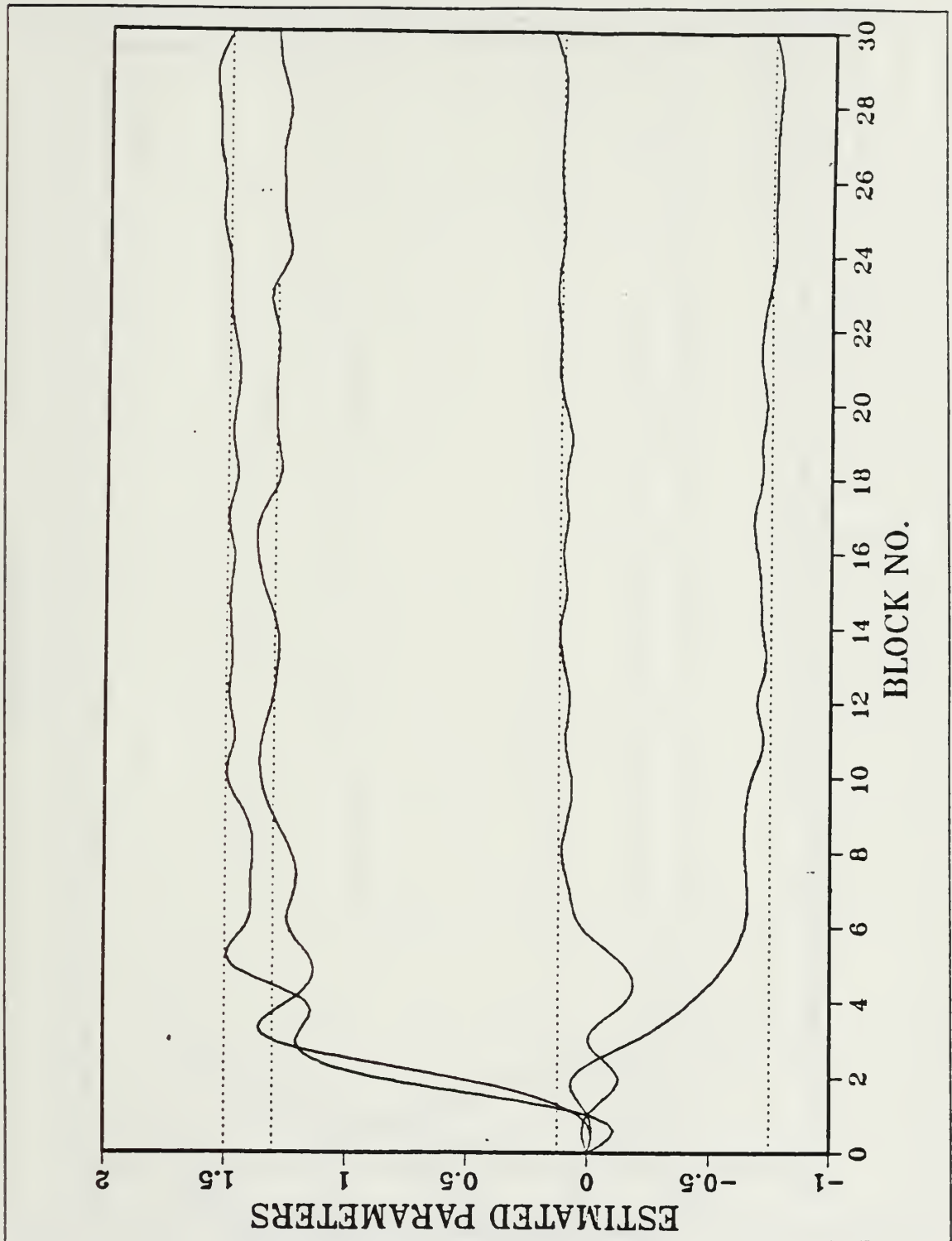Figure 4.13   Floating Point, With Noise, N = 15.

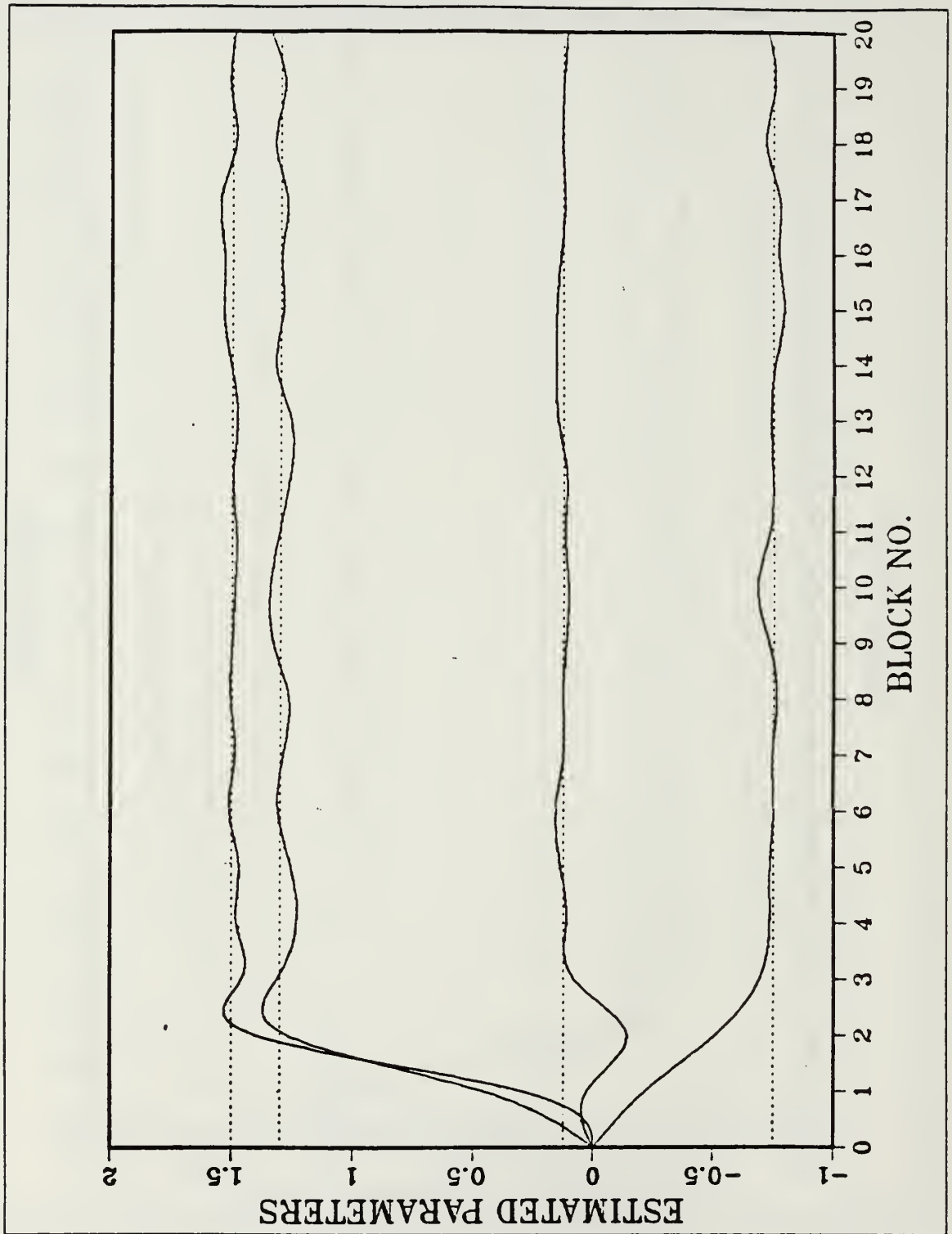Figure 4.14   Fixed Point, With Noise, N = 3.

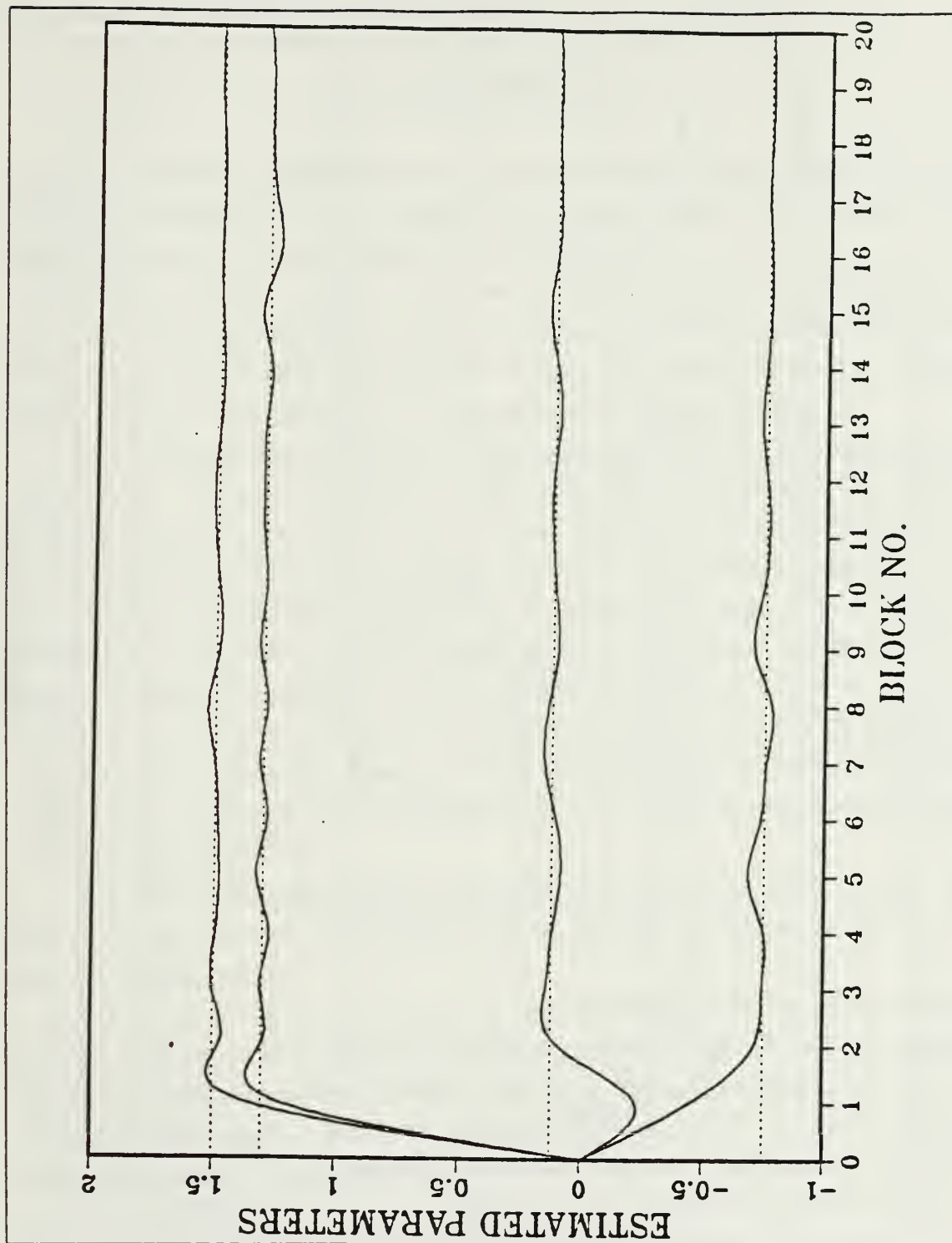Figure 4.15   Fixed Point, With Noise, N = 5.

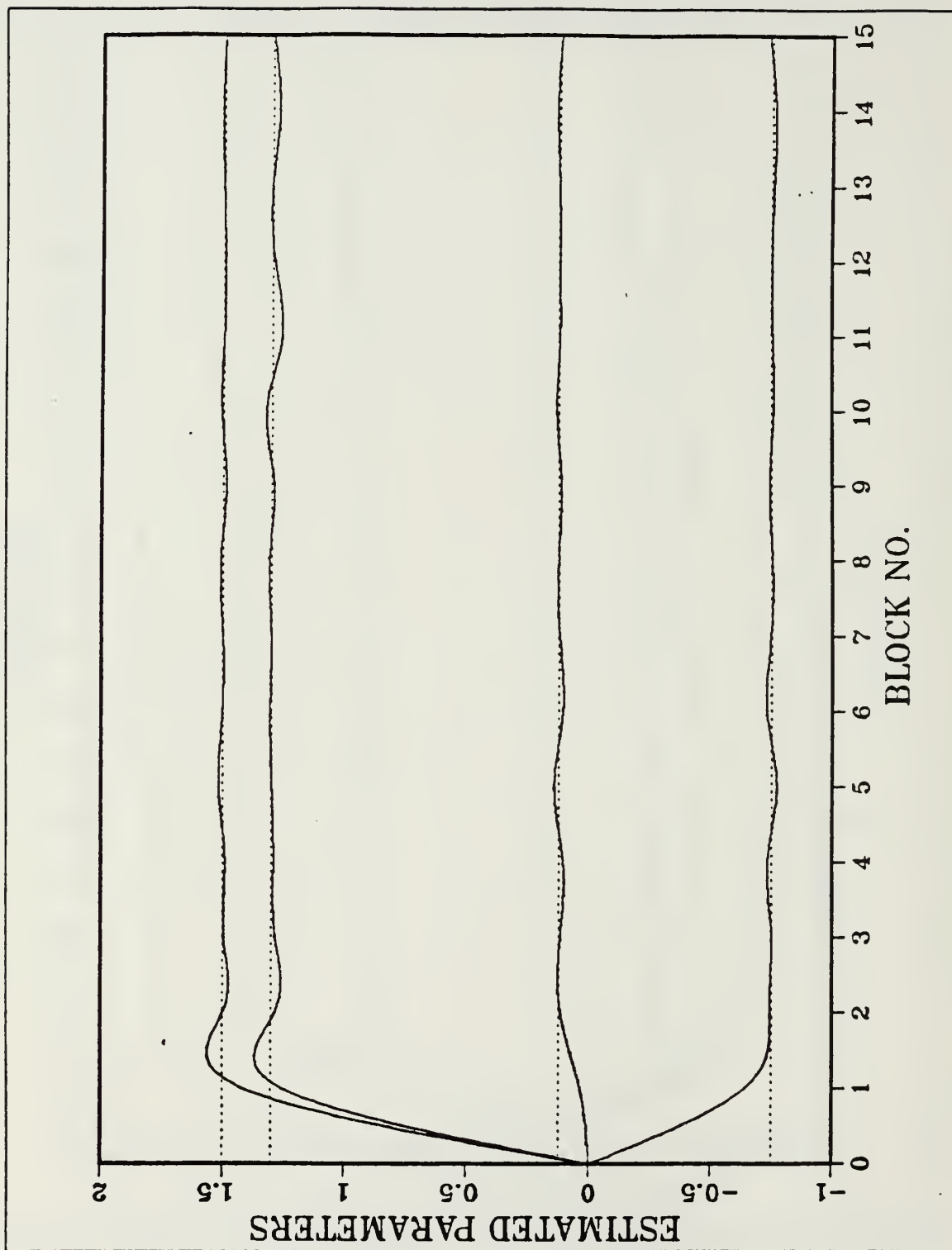Figure 4.16   Fixed Point, With Noise, N = 10.

57

Figure 4.17   Fixed Point, With Noise, N = 15.

V. CONCLUSIONS

The estimation of parameters using a parallel structure has been described in the preceding chapters. In this chapter, we discuss tradeoffs, advantages, and disadvantages of the various systems we have investigated.

Several possible conditions have been simulated in order to investigate the behavior of the parallel algorithm. We saw that the parameters converged well under all conditions, even in the presence of noise. Use of block processing tends to average out the effects of noise, perhaps at the slight expense of convergence rate.

A parallel algorithm suitable to hardware implementation has been presented in this research. The main contribution which distinguishes this algorithm from others available in the literature is the fact that we have replaced two different processors by two identical ones. With previous designs, a total of four different computing cells were required: two for the triangular section and two for the linear section. For the new design, we see that we need only two types of cells. These are the edge and boundary cells, whose operations were described in Chapter III. However, this new system also needs more total cells. Specifically, it can be shown that an additional $\frac{1}{2}d(d+1)$ cells are necessary, where d = dimension of system. Even for a sixth order system (d = 6), this requires only 21 additional cells. In a VLSI scheme, additional cells are considered inexpensive.

The second triangular section is somewhat slower than the linear section. We saw in Chapter III that (d-1) additional clock cycles were required to operate the second triangular section.

The functions of the cells must also be considered. The use of the Givens rotation by Equation (3.2) requires the use of a processor which can perform squaring operations, obtain square roots, multiply, divide, and do simple add and subtracts. Use of the CORDIC technique greatly simplifies the operations, in the sense that rotations can be implemented by use of addition and shifts only. Note from Table 1 that after rotation, the vector has increased in magnitude by the amount listed in the column k_i. This is easily compensated for by performing a right shift (division by 2) every other rotation or so.

Another important consideration is the use of fixed point versus floating point arithmetic. We saw in the simulation that the simple fixed point processors perform equally as well as floating point processors.

When the results of this thesis are considered, we see that a systolic array using the CORDIC technique presents an attractive means of implementing the parallel algorithm to identify unknown system parameters. This, in turn, leads to important applications in adaptive control systems and real time identification problems.

PROGRAM LISTING FOR SYSTOLIC ARRAY SIMULATION

This program calculates the parameter estimates for a discrete time linear system. It is written in Waterloo Pascal. In accordance with Pascal syntax rules, the size of the arrays must be defined in the declarations. Hence, one must know the order of the system to be modeled before attempting to use this program.

```pascal
program systolic  (input,output);

const
    dimension = 4; plusone = 5;  (* dimension is order of system *)
    gammacount = 30;   (* anticipated max number of rotations *)
    timerstop = 15;    (* used as index in main *)
type
    gamma_vector = record
        pi: boolean;
        scale_factor: real;
        params: array (.0..gammacount.) of integer;
        end;
    gamma_array = array (.1..dimension, 1..plusone.) of gamma_vector;
    theta_array = array (.1..dimension.) of real;
    u_array = array (.0..dimension, 1..plusone.) of real;
    a_array = array (.1..dimension, 1..plusone.) of real;

var
    gamma: gamma_array;                         (* direction of rotations *)
    gamma2: gamma_array;                        (* direction of rotations *)
    a: a_array;
    a2: a_array;
    theta: theta_array;
    u: u_array;
    u2: u_array;
    i,j,k,block_length,count,index: integer;
    timer: integer;      (*  timer variable  *)
    sigma: real;
    ch: char;
    h,m: text;



(*************************************************************************)
Procedure initialize (var sigma: real; var block_length:integer);
var
    i,j: integer;
begin
    for i := 1 to dimension  do
        for j:= 1 to plusone do
            a2(.i,j.)  := 0;
    writeln; writeln(m);
    writeln ('Initialize the systolic array');
    writeln (m,'Initialize the systolic array');
    writeln ('Choose the value of sigma'); writeln;
    writeln (m,'Choose the value of sigma'); writeln(m);
    readln (sigma); writeln (sigma); writeln(m,sigma);
    writeln; writeln(m);
    writeln ('Now enter the initial estimates of');
```

```
                writeln (m,'Now enter the initial estimates of');
                writeln ('the vector theta in order requested'); writeln;
                writeln (m,'the vector theta in order requested'); writeln(m);
                for i := 1 to dimension do
                    begin
                    writeln ('Theta ',i,' = ');
                    writeln (m,'Theta ',i,' = ');
                    readln (a(.i,dimension + 1.)); writeln; writeln(m);
                    writeln (a(.i,dimension + 1.)); writeln;
                    writeln (m,a(.i,dimension + 1.)); writeln(m);
                    end;
                writeln ('Enter the block length desired');
                writeln (m,'Enter the block length desired');
                readln (block_length);
                writeln (block_length:4);
                writeln (m,block_length:4);
                for i := 1 to dimension do
                    for j := 1 to dimension do
                        a(.i,j.) := 0;
                for i:= 1 to dimension do
                    begin
                    a(.i,i.) := sigma * 1;
                    write (m,a(.i,dimension + 1.):9:4);   (*write initial values*)
                    a(.i,dimension + 1.) := sigma * a(.i,dimension + 1.);
                    end;
                i := 0;
        end;
        (**************************************************************)


        (**************************************************************)
        Procedure reinitialize (sigma: real);
        var
            i,j: integer;
        begin
            for i := 1 to dimension  do
                for j:= 1 to plusone do
                    a2(.i,j.) := 0;
            for i := 1 to dimension do
                a (.i, plusone.) := sigma * theta (.i.);
            for i := 1 to dimension do
                for j := 1 to dimension do
                    a (.i,j.) := 0;
            for i := 1 to dimension do
                a(.i,i.) := sigma * 1;
        end;
        (**************************************************************)



        (**************************************************************)
        Procedure internal (x:real; var y,yout:real; var gammain, gammaout:
                                                    gamma_vector);

            (* This procedure performs the rotation on the x-y pair, given the
                gamma vector which contains the number and directions in which
                to rotate.  The new values of the x-y pair are passed out *)


        var
            i: integer;
            old_x, old_y: real;


            Procedure scale (var x,y: real; magfactor: real);
            begin
                x:= x * magfactor;
```

62

```pascal
            y:= y * magfactor;
        end;


    Function two (i:integer):real;
        (* This function calculates the exponentiation of the integer 2 to
           the power ( -i )  *)

    var
        n: integer;
        r: real;
    begin
        r := 1;
        if i = 0 then two := 1
        else for n := 1 to i do
            r := r * (1/2);
        two := r;
    end;

begin
    if gammain.pi then
        begin  x:= -x; y:= -y; end;
    old_x := x; old_y := y;
    i := 0;
    while (gammain.params(.i.) <> 9) and (i < gammacount) do
        begin
        if gammain.params(.i.) <> 0 then
            begin
            x := old_x + gammain.params(.i.) * two(i) * old_y;
            y := old_y - gammain.params(.i.) * two(i) * old_x;
            old_x := x; old_y := y;
            end;
        i := i + 1;
        end;
    scale (x,y, gammain.scale_factor);
    yout := y; y := x;
    gammaout := gammain;
end;
(*****************************************************************)


(*****************************************************************)
Procedure edge (x:real; var y: real; var gamma: gamma_vector);

    (* This procedure builds the gamma vector based upon the values of
       the x-y pair. The gamma vector contains only the values -1, 0, 1,
       or 9. If -1, then the vector is to be rotated counterclockwise.
       If +1, then it must be rotated clockwise. The value is set to
       zero if the next rotation would cause the new absolute value of
       y to be greater than the previous absolute value of y.  This way,
       we can prevent the rotation from taking place and cause the
       values to converge quickly.  The value of 9 is placed into the
       gamma vector once a pre-determined lower limit of y is reached,
       and it signals that no more rotations are to take place. *)
const
    low_limit = 1e-6;
var
    i : integer;
    temp_x, temp_y: real;
```

```pascal
Function two (i:integer):real;
    (* This function calculates the exponentiation of the integer
        2 to the power ( -i )  *)

var
    n: integer;
    r: real;
begin
    r := 1;
    if i = 0 then two := 1
    else for n := 1 to i do
        r := r * (1/2);
    two := r;
end;


Procedure one_rotation (x,y: real; var temp_x, temp_y: real);
        (* This procedure is used to do a rotation on the x-y pair.
            The input values of x and y are not changed, but the
            rotated values are passed out as temp_x and temp_y.     *)
var
    temp_gamma: integer;
begin
    if y > low_limit then temp_gamma := 1
    else temp_gamma := -1;
    temp_x := x + temp_gamma * two(i) * y;
    temp_y := y - temp_gamma * two(i) * x;
end;


begin
    i := 0;
    if x < 0 then begin                         (* initialize if x < 0  *)
        x := -x; y := -y; gamma.pi := true; end
    else gamma.pi := false;
    gamma.scale_factor := 1;                     (* initialize scale factor *)
    while (abs(y) > low_limit) and (i < gammacount) do
        begin
        one_rotation (x,y,temp_x, temp_y);(* check to see what rotated *)
                                          (* values of x,y will be *)
        while (abs(y) < abs(temp_y)) and (i < gammacount) do
            begin                         (* repeat this loop until new *)
            gamma.params(.i.) := 0;       (* y < old y                  *)
            i := i + 1;
            one_rotation (x,y,temp_x, temp_y);
            end;
        if y > low_limit then gamma.params(.i.) := 1 (*do CW rotation *)
        else if y < -low_limit then gamma.params(.i.) := -1;  (* CCW  *)
        i := i + 1;
        x := temp_x; y := temp_y;         (* update the rotated values*)
        if y > low_limit then gamma.params(.i.) := 1
        else if y < -low_limit then gamma.params(.i.) := -1;
        if i = 1 then gamma.scale_factor:= 1/sqrt(2) (* update scaling *)
        else gamma.scale_factor:= gamma.scale_factor *
                                    cos(arctan(two(i-1)));
        end;            (* while *)
    gamma.params(.i.) := 9;                      (* end of rotations *)
    y := x * gamma.scale_factor;                 (* scale final values *)
end;
(***************************************************************************)
```

```
begin                              (* main *)
    reset (h, 'data text a1');
    rewrite(m, 'outfile text a1');
    initialize (sigma, block_length);
    for timer := 1 to timerstop do
        begin
        for index := 1 to block_length do
            begin
            for j := 1 to dimension + 1 do
                read (h,u(.0,j.));
            readln (h);
            for k := 2 to 2 * dimension + 1 do
                begin
                for i := 1 to dimension do
                    begin
                    j := k - i;
                    if (i <= j) and (j <= dimension + 1) then
                        begin
                        if i = j then
                            edge (u(.i-1,j.), a(.i,j.), gamma(.i,j.))
                        else
                            internal (u(.i-1,j.), a(.i,j.), u(.i,j.),
                                     gamma (.i,j-1.), gamma(.i,j.));
                        end;              (* if i < j *)
                    end;                  (* for i = 1 to dimension   *)
                end;                      (* for k *)
            end;                          (* for index *)


                (*   solve system *)

    a2(.1,1.) := a(.dimension, dimension.);        (* shift bottom of *)
    a2(.1,plusone.) := a(.dimension, plusone.);    (* a into a2       *)
    theta (.dimension.) := a2(.1,plusone.) / a2(.1,1.);
    count := dimension - 1;
    while count >= 1 do
        begin
        for j := dimension downto 2 do          (* shift down a matrix *)
            for k := 1 to plusone do            (* and feed into u     *)
                a(.j,k.) := a(.j-1,k.);
        for j := 1 to dimension do
            u2(.0,j.) := a(.dimension, plusone - j.);
        u2(.0,plusone.) := a(.dimension, plusone.);
        for k := 2 to 2 * dimension + 1 do
            begin
            for i := 1 to dimension do
                begin
                j := k - i;
                if (i <= j) and (j <= dimension + 1) then
                    begin
                    if i = j then
                        edge (u2(.i-1,j.), a2(.i,j.), gamma2(.i,j.))
                    else
                        internal (u2(.i-1,j.), a2(.i,j.), u2(.i,j.),
                                 gamma2(.i,j-1.), gamma2(.i,j.));
                    end;          (* if i < j *)
                end;              (* for i = 1 to dimension   *)
            end;                  (* for k *)
        theta (.count.) := a2 (.dimension - count + 1, plusone.) /
                a2 (.dimension - count + 1, dimension - count + 1.);
        count := count - 1;
        end;      (* while *)
    for i := 1 to dimension do              (* write out values     *)
        write (m, theta (.i.):9:4);
    reinitialize (sigma);
    end;          (* for timer *)
end.
```

65

LIST OF REFERENCES

1. Goodwin, G.C. and Sin, K.S., *Adaptive Filtering, Prediction and Control*, Prentice-Hall, 1984.

2. Kim, Yong Hong, *A Parallel Structure for On Line Identification and Adaptive Control*, Masters Thesis, Naval Postgraduate School, Monterey, California, March 1987.

3. Kung, H.T. and Leiserson, Charles E., "Systolic Arrays (for VLSI)," *Sparse Matrix Proceedings*, SIAM, 1978.

4. Stewart, G. W., *Introduction to Matrix Computations*, Academic Press, 1973.

5. Haykin, Simon, *Adaptive Filter Theory*, Prentice-Hall, 1984.

6. Ljung, Lennart, and Söderström, Torsten, *Theory and Practice of Recursive Identification*, MIT Press, 1983.

7. Hsia, T.C., *System Identification*, Lexington Books, 1977.

8. Schmid, H., *Decimal Computation*, John Wiley and Sons, 1974.

BIBLIOGRAPHY

Anton, Howard, *Elementary Linear Algebra*, John Wiley & Sons, 1984.

Cristi, Roberto, *A Parallel Structure for Adaptive Pole Placement Algorithm*, to be published, March 1986.

Jennings, Alan, *Matrix Computation for Engineers and Scientists*, John Wiley & Sons, 1977.

Noble, Ben and Daniel, James W., *Applied Linear Algebra*, Prentice-Hall, 1977.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Prof. Roberto Cristi, Code 62Cx 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5002

4. Prof. Sherif Michael, Code 62Mi 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5002

5. Major Richard Adams, Code 52Ad 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5002

6. Paul A. Willis 2
   17120 Via Pasatiempo
   San Lorenzo, CA 94580

7. Mr. V. V. Saba 1
   717 Carlin Ct.
   Carmel, IN 46032

8. Mr. Joseph Smulewicz 1
   5172 Sabin Ave.
   Fremont, CA 94536

9. Dr. K. Bromley,
   Code 74It
   NOSC
   San Diego, CA 92152 1

10. OP - 02
    Undersea Warfare
    Washington, D.C. 20350 1